拨云见日

基于Android的内核与系统架构源码分析



王森◎著

清华大学出版社

拨云见日

基于Android的内核与系统架构源码分析



王森◎著

清华大学出版社
北京

内 容 简 介

本书包括上下两篇内容。上篇在保证完整 Linux 内核架构分析的前提下，着重分析 Android 系统中强烈依赖的 Linux 内核机制，如多核 ARM 架构的支持，而略去 Android 系统产品化没有用到内核机制，如 SWAP 机制。下篇主要分析 Android 系统层主要架构机制，尤其注重分析这些用户态机制与内核机制的接驳与交互。本书整理自作者多年积累的笔记，形式以源代码分析为主。

本书适合相关领域工程师作为实际项目的参考，以及有志于通过研读源码掌握 Android 系统与 Linux 内核精髓的读者。

图书在版编目（CIP）数据

拨云见日——基于 Android 的内核与系统架构源码分析 / 王森编著. —北京：清华大学出版社，2015
ISBN 978-7-302-38199-0

Ⅰ. ①拨…　Ⅱ. ①王…　Ⅲ. ①移动终端-应用程序-程序设计　Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字（2014）第 230612 号

责任编辑：夏兆彦
封面设计：
责任校对：徐俊伟
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn, http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张：**27　　**字　　数：**668 千字
版　　次：2015 年 1 月第 1 版　　**印　　次：**2015 年1 月第 1 次印刷
印　　数：
定　　价：　　元

产品编号：056459-01

本书推荐语

对于像 Android、Linux 这样复杂的大型软件，泛泛了解一下工作原理并不难，看看文档，或者随便找几本书看看，再实际用一下体验体验，也就差不多了。

但是若想要有比较深入透彻的理解，那就非得要阅读分析其源代码不可，这就比较难了。这时候，如果有人先行一步，下了苦功先把它弄懂，再把所获的知识和心得分享出来，那就是很有价值了。我觉得王森这本书就是这样，里面不光有代码的分析，还有他的见解，特别是还有他的一些经验之谈，相信读者会和我一样看了后觉得受益匪浅。

——《Linux 内核源代码情景分析》作者 毛德操

Android 系统几年来高速发展，不过国内工程师使用者居多，深入了解者少。本书深入浅出，将作者多年实践经验结合系统深层探索呈现给大家，可谓雪中送炭、广大 Android 工程师的福音。感谢本书作者的分享，相信本书会成为工程师们进阶修炼的极大助力。

——北京时代飞腾科技有限公司 技术总监 刘天宇

本书内容风趣幽默且比喻贴切，“Android 与 Linux 内核的关系，就好比一辆整车和底盘发动机的关系”；该书内容详实而又丰富，既有 MTK 平台的知识，又有三星平台的细节，既有 Linux 基于 ARM 的实现细节，又有 Android 的原理和机制。作者苦心钻研，颇有心得，深入浅出又提纲挈领地将 ARM 和 Android 的核心机密全盘托出，相信此书定会给无论是初学者还是资深开发者带来帮助和益处。

——三星半导体杭州研究所资深软件工程师 张秀文

本书在着力分析 Android 系统最常用到的内核机制之后，继续向上剖析 Android 用户层核心机制是如何接驳 Linux 内核的。而且书中分析涉及到 ARM 体系为较新 Cortex A9 SMP 架构，对于读者开发、研究工作有着实际的借鉴作用。

——红狼软件创始人；《深入剖析 Android 系统》作者 杨长刚

记得当年从研一开始，本书作者就埋头于 Linux 源代码的学习、分析，梦想着了解 Linux 的每一个角落。虽然他常常不修边幅，但只要谈起编程或 Linux 源代码他就两眼放光，滔滔不绝地大谈特谈心得体会。那时他的执着就让我十分钦佩。一晃十多年过去了，本书作者仍旧是那个严肃的程序员，依旧在执着地追求着他的梦想。

如果你也是一位严肃的程序员，有兴趣，有毅力去了解 Android 的实现；或者你是一个 Android 系统开发人员，推荐把本书作为手边的参考书。源代码注释的写作方式看起来有些简陋，但是这样对阅读 Android 源代码非常有帮助，根据书中的代码段可以很容易地搜索到你关注的 Android 源代码，从而大大缩短学习时间。

——本书作者同学；CA Technologies 研发经理 王晋强

前　　言

1996 年，面对闪烁的 DOS 提示符，笔者心中产生了一个愿望——能够理解计算机每一条指令是如何工作的。

2000 年，笔者明白一个道理，内核才是机器的灵魂，只有阅读源码才能与机器的灵魂对话。于是笔者开始了漫长的源码阅读之旅。

2008 年，在经历了近十年内核以及驱动项目开发后，笔者发现尽管玄机重重，但是内核并非不能驾驭。在浩如烟海的 Linux 代码里有一个清晰的内核骨架，Linux 各种文件、驱动子系统围绕该骨架接驳起来，把握住这个骨架就驾驭了 Linux。

然而痛苦随之而来，以前笔者以为内核就是一切，驾驭了内核就理解了机器的一切。但是实际上，内核不是全部，要理解机器的机理还需要操作系统领域的探索，在经历了 Glibc、X、GTK 等痛苦且失败的探索之后，笔者发现了 Android。

2011 年，笔者多年的心愿总算有个了结。在经历了 2 年多 Android 源码研究之后，终于理解了 Android 如何将内核的强大能量释放给应用程序，如何对传统内核加以巧妙改造以满足当今以面向对象机制为基础的软件体系。这一年又是新的开始，一次难得的机会，笔者能够深入硬件开发。

2013 年，这两年可谓笔者的硬件生涯。无论是原理图、PCB 设计还是选择制板工厂、实施备料、贴片试产跟线，笔者都能够有幸深入一线体验硬件工程的复杂与艰辛。笔者尝试了软硬件协同设计，尽管在板级设计这一层面作用有限，但是通盘软考虑依然可以避免不少工程误区。

2014 年，笔者将这些年的笔记进行整理，攒成此书，也算是个人生的小结吧。

本书其实是实现本人年少时梦想的一部分——理解计算机每一条指令是如何工作的。

然而，这只是一个梦想，操作系统如同浩如烟海的原始森林，每一条指令只是其中的一片树叶，人们不可能读完每一行代码，也没有必要这样做。任何一个优秀的操作系统都有一个精悍的架构，Android 系统也不例外，本书通过分析这个架构来阐述 Android 机理。

笔者认为，尽管 Android 用户层面的代码量远大于内核部分，但是其关键架构却与内核息息相关，且其功能部分架构已经有很多优秀书籍和资料阐述。所以本书更多着墨于 Android 内核表现以及内核机制的 Android 运用。

最后需要提醒读者的是，无论多丰富的语言在源码面前也是苍白的，本书选择通过源码注释的方式来描述 Android 架构，建议读者结合源码来阅读本书。

编者
2014 年 4 月

目　　录

上篇　内核

上篇　内核

Android 与 Linux 内核的关系，就好比一辆整车和底盘发动机的关系。作为底盘和发动机的 Linux 内核，尽管不能直接被用户和应用开发者感知到，但是却决定了 Android 系统运行时最核心的机制。

第 1 章　ARM 多核处理器

计算性能是处理器演进的第一动力。然而，尽管各种架构的高性能处理器层出不穷，但真正大规模普及开来的似乎只有 Intel 和 ARM 体系。观察其中的现象不难发现如下规律。

虽然处理器性能得到大幅改善，但如果无法得到现有主流操作系统的支持，就无法大规模应用。进一步来讲，即使某种处理器得到主流操作系统的支持，但是由于其指令集的不兼容性，导致大量的应用无法运行，这种处理器也是难以普及的，安腾的失败是个很好的例子。

“兼容性”是处理器演进的第一法则。现有的软件体系是计算机世界的主宰，所有不服从现在软件体系的指令集都只能被边缘化或者被淘汰。所以看到 Intel 和 ARM 不断扩展寄存器长度、增加发射单元、支持乱序、扩展总线单元等一系列手段来改进处理器架构，就是不敢废弃一条既有指令。

降低计算功耗是处理器演进的第二动力。对于服务器来说功耗就是运营成本，对于移动计算来说功耗就是生命。似乎台式机功耗要求不大，但是台式机与服务器、移动设备不仅在处理器的生产及设计上共享基础设施外，软件体系也有着相同的基础，导致整个软件体系为了照顾移动设备和服务器而收敛了扩张的步伐。这就是人们常说的“计算过剩”，其实计算永远不会过剩，只是庞大的软件体系有所收敛罢了。

频率提升是功耗的大敌，为了提高能耗比，设计频率更低而核心更多的处理器是好的方法，由此，近年来处理器的扩张由单核高频转变为多核体系。

Linux 支持的多核体系的基本特点为：所有处理器拥有完全相同的指令集，所有处理器共享同一内存。如果不符合以上任何一点，操作系统内核都需要做架构及修改才能支持。若满足这两个前提，其他方面的问题，都可以通过内核和系统小规模修改来支持。如 Big.little 架构下互相搭配的处理器尽管核心内部实现不同，但各核心有着相同的指令集，Linux 内核可以将它们当作同样的处理器来分配线程。内核在完全不加改动的情况下，也许会出现一个重量级线程被分配到了一个 little 的核心上，但是内核本身可以通过处理器负载均衡将其平衡到别的处理器上。当 little 负载较高时，big 必定会上线运行，但是系统有可能会出现重负载线程独自霸占 little 的情形的极限情况，这时需要将调度算法稍加修改，首先记录每个处理器能力，然后监测 little 上单线程高负载发生的时长，若超出一定阀值则将高负载线程时间片减为零，从 little 摘下来，再将其挂到 big 处理器运行队列即可。

进一步讲，多核心之间只要基本的读写、跳转之类指令相同，其余指令集不同，Linux 也可以支持的。内核只需用到基本指令即可，各核心用户空间指令可以不同。内核需做如下修改：每颗核心的调度队列记录下其核心用户态指令集特性，每个线程创建之初申明自身需要的指令集特性，内核在给线程分配核心及做负载均衡时比较两者是否匹配。用户层动态加载器也要做修改，针对当前处理器指令集特性动态加载、跳转到不同版本的动态库中，且在动态库执行过程中标记自身线程不可切换处理器。当然这种复杂情况超出了当前

现状，尽管有用户态指令集不同的架构，但是运行指令集差集的线程不需要动态加载库的支持，且直接绑定到指定核心上。

1.1　SMP 相关基础数据结构

CPU 管理的特点是自我管理，除了在启动、休眠、调频受控于 CPU0 的工作以外，处理器相关的绝大部分工作都由处理器自我管理。处理器是内核的执行体，又被内核控制。内核中准备了表征处理器运行状态的相应数据结构，处理器在运行时将自己的状态记录在这些结构中，而处理器也能通过别的处理器的表征结构了解其他处理器状态或发起控制。

每颗 CPU 都对应一个通用 CPU 结构，将 ARM Core 与 device 联系起来。

```
struct cpu {
/*cpu 编号*/
    int node_id;
/*对于 arm smp 该值为真，表示每颗 cpu 都可以被关掉*/
    int hotpluggable;
/*正如每个外设都有一个 struct device 表示自身一样，cpu 也不例外*/
    struct device dev;
 };
```

处理器在内核中也被作为系统设备存在，而其相关操作以驱动形式通过处理器系统设备类——struct sysdev_class cpu_sysdev_class 来识别管理处理器。在处理器拓扑初始化时，内核完成设备侧的注册。

```
//处理器拓扑初始化
static int __init topology_init(void)
{   …
   /*对于每个物理存在处理器的操作，位图 cpu_possible_bits 描述了系统中的处理器，无论处理器是否 online，都对应其中一位*/
    for_each_possible_cpu(cpu) {
        struct cpuinfo_arm *cpuinfo = &per_cpu(cpu_data, cpu);
        /*ARM SMP 的架构是每个 core 都可以被单独关闭的*/
        cpuinfo->cpu.hotpluggable = 1;
        /*将当前 cpu 的 struct device 注册到 struct sysdev_class cpu_sysdev_class*/
        register_cpu(&cpuinfo->cpu, cpu);
    }
    …
}

//注册一颗处理器的设备
int __cpuinit register_cpu(struct cpu *cpu, int num)
{
    …
```

```
        //对 CA9，非 NUMA，为 0
        cpu->node_id = cpu_to_node(num);
        cpu->sysdev.id = num;
        //处理器系统设备类
        cpu->sysdev.cls = &cpu_sysdev_class;
        /* 向处理器系统设备类注册该处理器，在每注册一个设备时，都会调用设备类驱动来初始化
        该驱动*/
        error = sysdev_register(&cpu->sysdev);
        …
    }
```

在对方驱动侧，通过向处理器系统设备类注册驱动来匹配初始化处理器，参见 1.3 节。

struct cpuinfo_arm 对应于每颗 Arm core 实体，记录其基本信息，在 ARM 初始化时会依次初始化并注册每个 struct cpuinfo_arm。

```
struct cpuinfo_arm {
    struct cpu  cpu;
#ifdef CONFIG_SMP
    //当前 cpu 的 idle 线程
    struct task_struct *idle;
    unsigned int    loops_per_jiffy;
#endif
};
```

CPU 设备集合，每个 CPU 的 struct device 和 struct sysdev_driver 都会被加进来。这是联系 CPU 设备集合类驱动的纽带，当 CPU 设备或驱动注册的时候，会依次扫描这里的驱动列表或 CPU 设备，并进行初始化，如 cpufreq_sysdev_driver。

```
struct sysdev_class cpu_sysdev_class = {
    .name = "cpu",
    .attrs = cpu_sysdev_class_attrs,
};
```

struct cpumask：CPU 状态的基本数据结构定义如下：

```
 typedef struct cpumask { DECLARE_BITMAP(bits, NR_CPUS); } cpumask_t;
#define DECLARE_BITMAP(name,bits) \
    unsigned long name[BITS_TO_LONGS(bits)]
```

展开如下：

```
typedef struct cpumask { unsigned long bits[BITS_TO_LONGS(NR_CPUS)] }
cpumask_t;
```

cpu_possible_mask 位图，用来表示系统中的 CPU，每颗处理器对应其中一位。

cpu_online_mask 位图，用来表示当前处于工作状态的 CPU，每颗处理器对应其中一位。

```
cpu_bit_bitmap:
```

```
const unsigned long cpu_bit_bitmap[BITS_PER_LONG+1][BITS_TO_LONGS(NR_CPUS)]
```

对于双核 CA9，这个数组是 const unsigned long cpu_bit_bitmap[33][1]。可见这是列数为 1 的数组。其中列数与 CPU 个数和 sizeof(long)有关。每颗 CPU 对应一行，但是最上面的一行是保留不用的。

```
//接下来是该位图具体的定义，通过一组宏来实现
const unsigned long cpu_bit_bitmap[BITS_PER_LONG+1][BITS_TO_LONGS(NR_
CPUS)] = {
    MASK_DECLARE_8(0),  MASK_DECLARE_8(8),
    …
    MASK_DECLARE_8(48), MASK_DECLARE_8(56),
};
```

CPU0～CPU7 对应于前 8 个数据，其宏定义为 MASK_DECLARE_8(0)，接下来以此为例，通过相关宏定义的层层替代，分析处理器位图结构。

第 1 步，展开顶层宏：

```
#define MASK_DECLARE_8(x)   MASK_DECLARE_4(x), MASK_DECLARE_4(x+4)
```

得到如下第二层宏数组：

```
MASK_DECLARE_4(0), MASK_DECLARE_4(0+4)
```

第 2 步，将宏定义：

```
#define MASK_DECLARE_4(x)MASK_DECLARE_2(x), MASK_DECLARE_2(x+2)
```

展开，得到第三层宏数组：

```
MASK_DECLARE_2(0),MASK_DECLARE_2(0+2);MASK_DECLARE_2(4),MASK_DECLARE_2(4+2)
```

第 3 步，将宏定义：

```
#define MASK_DECLARE_2(x)MASK_DECLARE_1(x), MASK_DECLARE_1(x+1)
```

展开，得到第四层宏数组：

```
MASK_DECLARE_1(0), MASK_DECLARE_1(0+1);
…
MASK_DECLARE_1(6), MASK_DECLARE_1(6+1)
```

最后，再将宏定义

```
#define MASK_DECLARE_1(x)
```

替换为

```
[x+1][0] = (1UL << (x))
```

得到如下序列：

```
[1][0] = (1UL << (0))
[2][0] = (1UL << (1))
```

```
⋮
[8][0] = (1UL << (7))
```

由此可见，第 0 行没有初始化。对于 CPU0、CPU1，其 MASK 分别对应第 1 行和第 2 行，值分别为(1UL << (0))，(1UL << (1))。

而对于 CPU8～31，则由 MASK_DECLARE_8(8)、MASK_DECLARE_8(16)、MASK_DECLARE_8(24)定义了对应的 MASK 了，其过程与 MASK_DECLARE_8(0)类似，这里不做展开。

1.2　Percpu 内存管理

随着处理器核心的增加，内核中系统中并发的线程也随之增加，这样对一些共享数据同时访问的几率也就增加，就避免不了使用 spin_lock，而且往往处理器核心越多造成的麻烦越大。Percpu 内存对这种数据无能为力，但是内核中有些数据只是处理器局部，可见，这种数据不会被别的处理器访问到，不需要加以 spin_lock 而直接访问。Percpu 内存适用于这种数据，而实际上处理器局部数据的实际分配还是通过内核基本 slab 进行，而 Percpu 内存则是用来存放指向处理器局部数据的指针。

1.2.1　内核显式定义的处理器局部数据

这是在内核代码中手工定义的变量，但是这些数据比较特殊，内核定义与引用时使用特殊的宏，链接时被集中排放在特殊的地址上，内核初始化时有着专门的指针处理。本节相关分析围绕这些方面展开。

首先分析链接文件 arch/arm/kernel/vmlinux.lds.S 中定义的这些特殊地址。

展开宏：PERCPU_SECTION(32)

```
#define PERCPU_SECTION(cacheline)                          \
    . = ALIGN(PAGE_SIZE);                                  \
    .data..percpu   : AT(ADDR(.data..percpu) - LOAD_OFFSET) {   \
        VMLINUX_SYMBOL(__per_cpu_load) = .;                \
        PERCPU_INPUT(cacheline)                            \
    }

#define PERCPU_INPUT(cacheline)                            \
    VMLINUX_SYMBOL(__per_cpu_start) = .;                   \
    *(.data..percpu..first)                                \
    . = ALIGN(PAGE_SIZE);                                  \
    *(.data..percpu..page_aligned)                         \
    . = ALIGN(cacheline);                                  \
    *(.data..percpu..readmostly)                           \
    . = ALIGN(cacheline);                                  \
    *(.data..percpu)                                       \
    *(.data..percpu..shared_aligned)                       \
    VMLINUX_SYMBOL(__per_cpu_end) = .;
```

该宏的作用是声明了以.data..percpu..first、.data..percpu..readmostly、.data..percpu 等为名字的数据段，内核代码中如果有变量的属性指出放在该数据段的时候，链接时将其分配到对应的数据段。而且为了能够在内核里直接访问这段区域，定义两个变量：__per_cpu_start 和__per_cpu_end。

在内核编码时，对处理器局部数据的定义需使用特殊的宏（DEFINE_PER_CPU）来完成。这里分析（DEFINE_PER_CPU）的实现。

层层展开该宏：

```
#define DEFINE_PER_CPU(type, name)                       \
    DEFINE_PER_CPU_SECTION(type, name, "")

#define DEFINE_PER_CPU_SECTION(type, name, sec)          \
    __PCPU_ATTRS(sec) PER_CPU_DEF_ATTRIBUTES             \
    __typeof__(type) name

#define __PCPU_ATTRS(sec)                                \
    __percpu __attribute__((section(PER_CPU_BASE_SECTION sec)))    \
    PER_CPU_ATTRIBUTES

#define PER_CPU_BASE_SECTION ".data..percpu"
```

根据以上宏定义展开之：可以得到

```
__attribute__((section(.data..percpu))) __typeof__(type) name
```

可见宏 DEFINE_PER_CPU(type, name)的作用就是将类型为 type 的 name 变量放到.data..percpu 数据段中。

同样方法可以推出：宏 DECLARE_PER_CPU_READ_MOSTLY(type, name)的作用就是将类型为 type 的 name 变量放到.data..percpu..readmostly 数据段中。这种数据段是 cacheline 对齐，可以大大提高 cache 的利用率，很适合频繁读取数据，不过在笔者分析的这个内核版本 3.0.13 中，内核中这种数据还没有大规模使用的。而且 cacheline 定义为 32，对于不同架构可以使用时适当调整。

另外宏 DECLARE_PER_CPU_FIRST(type, name)对应.data..percpu..first 数据段。

宏 DECLARE_PER_CPU_PAGE_ALIGNED 对应.data..percpu..first”数据段。

内核初始化过程中，会为每个处理器定位链接进镜像的局部数据。

```
//初始化是_ _per_cpu_offset 的建立
void_ _init setup_per_cpu_areas(void)
{
    rc = pcpu_embed_first_chunk(PERCPU_MODULE_RESERVE,
                   PERCPU_DYNAMIC_RESERVE, PAGE_SIZE, NULL,
                   pcpu_dfl_fc_alloc, pcpu_dfl_fc_free);

    delta = (unsigned long)pcpu_base_addr - (unsigned long)_ _per_cpu_start;
  /*设置每个 cpu 的_ _per_cpu_offset 指针*/
```

```
        for_each_possible_cpu(cpu)
            _ _per_cpu_offset[cpu] = delta + pcpu_unit_offsets[cpu];
}
```

再看对内核中对 percpu 数据的引用：内核通过宏 per_cpu 对定义的处理器局部数据的引用，展开该宏：

```
#define per_cpu(var, cpu) \
    (*SHIFT_PERCPU_PTR(&(var), per_cpu_offset(cpu)))

#define SHIFT_PERCPU_PTR(__p, __offset) ({              \
    __verify_pcpu_ptr((__p));                           \
    RELOC_HIDE((typeof(*(__p)) __kernel __force *)(__p), (__offset)); \
})

#define per_cpu_offset(x) (__per_cpu_offset[x])

/*选用一个简单易看得 RELOC_HIDE 宏定义*/
# define RELOC_HIDE(ptr, off)                           \
  ({ unsigned long __ptr;                               \
    __ptr = (unsigned long) (ptr);                      \
(typeof(ptr)) (__ptr + (off)); })
```

可见是以当前 CPU 局部数据__per_cpu_offset[x]为基地址的相对寻址。

1.2.2 Percpu 内存管理的建立

在建立 Percpu 内存管理机制之前要整理出该架构下的处理器信息，包括处理器如何分组、每组对应的处理器位图、静态定义的 Percpu 变量占用内存区域、每颗处理器 Percpu 虚拟内存递进基本单位等信息。本文仅以双核 CA9 处理器作为分析目标。

对于处理器的分组信息，内核使用 struct pcpu_group_info 结构表示：

```
struct pcpu_group_info {
    /*该组的处理器数目，对于双核 CA9 处理器，该值为 2 */
    int             nr_units;
    /*组内处理器数目×处理器 percpu 虚拟内存递进基本单位*/
    unsigned long       base_offset;
   /*组内处理器对应数组，双核 CA9 架构下该数组长度为 2*/
    unsigned int        *cpu_map;
};
```

整体的 Percpu 内存管理信息被收集在 struct pcpu_alloc_info 结构中：

```
struct pcpu_alloc_info {
    //静态定义的 percpu 变量占用内存区域长度
    size_t          static_size;
   /*预留区域，在 percpu 内存分配指定为预留区域分配时，将使用该区域*/
```

```
    size_t          reserved_size;
    size_t          dyn_size;
   /*每颗处理器的 percpu 虚拟内存递进基本单位*/
    size_t          unit_size;
    …
   /*该架构下的处理器分组数目，CA9 双核架构下该值为 1*/
    int         nr_groups;
/*该架构下的处理器分组信息，CA9 双核架构下该数组长度为 1*/
    struct pcpu_group_info  groups[];
};
```

接下来构建静态定义的 percpu 变量创建 percpu 区域：

```
/*该函数被 void __init setup_per_cpu_areas(void) 调用，对于 CA9 架构其参数
cpu_distance_fn 为 NULL，参数 alloc_fn 为 pcpu_dfl_fc_alloc*/
int __init pcpu_embed_first_chunk(size_t reserved_size, size_t dyn_size,
                 size_t atom_size,
                 pcpu_fc_cpu_distance_fn_t cpu_distance_fn,
                 pcpu_fc_alloc_fn_t alloc_fn,
                 pcpu_fc_free_fn_t free_fn)
{
    void *base = (void *)ULONG_MAX;
    void **areas = NULL;
    struct pcpu_alloc_info *ai;
    size_t size_sum, areas_size, max_distance;
    int group, i, rc;
   /*收集整理该架构下的 percpu 信息，结果放在 struct pcpu_alloc_info 结构中*/
    ai = pcpu_build_alloc_info(reserved_size, dyn_size, atom_size,
                  cpu_distance_fn);
    …
   //静态定义变量占用空间+reserved 空间+动态分配空间
   size_sum = ai->static_size + ai->reserved_size + ai->dyn_size;
    …
    /*针对每个 group 操作 */
    for (group = 0; group < ai->nr_groups; group++) {
        struct pcpu_group_info *gi = &ai->groups[group];
        unsigned int cpu = NR_CPUS;
        void *ptr;

        …

        /* 为该 group 分配 percpu 内存区域。长度为处理器数目×每颗处理器的 percpu 递进
        单位。函数 pcpu_dfl_fc_alloc 是从 bootmem 里取得内存，得到的是物理内存 */
        ptr = alloc_fn(cpu, gi->nr_units * ai->unit_size, atom_size);
        …
        areas[group] = ptr;
```

```
        base = min(ptr, base);
        //为每颗处理器建立其 percpu 区域
        for (i = 0; i < gi->nr_units; i++, ptr += ai->unit_size) {
            if (gi->cpu_map[i] == NR_CPUS) {
                /*检查组内处理器，对于没有用到的处理器释放其 percpu 区域 */
                free_fn(ptr, ai->unit_size);
                continue;
            }
            /*将静态定义的 percpu 变量拷贝到每颗处理器 percpu 区域*/
            memcpy(ptr, __per_cpu_load, ai->static_size);
            /*为每颗处理器释放掉多余的空间，多余的空间是指 ai->unit_size 减去静态
            定义变量占用空间+reserved 空间+动态分配空间*/
            free_fn(ptr + size_sum, ai->unit_size - size_sum);
        }
    }

    /* 处理器架构相关的计算，对于 CA9 双核架构，ai->nr_groups 为 1，且
ai->groups[group].base_offset 和 max_distance 这里的计算结果都为 0*/
    max_distance = 0;
    for (group = 0; group < ai->nr_groups; group++) {
        ai->groups[group].base_offset = areas[group] - base;
        max_distance = max_t(size_t, max_distance,
                     ai->groups[group].base_offset);
    }

    max_distance += ai->unit_size;
    …
    //建立可动态分配的 percpu 内存区域
    rc = pcpu_setup_first_chunk(ai, base);
    …
}


//建立可动态分配的 percpu 内存区域
int __init pcpu_setup_first_chunk(const struct pcpu_alloc_info *ai,
                 void *base_addr)
{
    static char cpus_buf[4096] __initdata;
    static int smap[PERCPU_DYNAMIC_EARLY_SLOTS] __initdata;
    static int dmap[PERCPU_DYNAMIC_EARLY_SLOTS] __initdata;
    …

    for (cpu = 0; cpu < nr_cpu_ids; cpu++)
        unit_map[cpu] = UINT_MAX;
```

```
    pcpu_first_unit_cpu = NR_CPUS;

    /*针对每一group的每一颗处理器,对于双核CA9,ai->nr_groups值为0,gi->nr_units
    值为2*/
    for (group = 0, unit = 0; group < ai->nr_groups; group++, unit += i)
{
        const struct pcpu_group_info *gi = &ai->groups[group];

       //该组处理器的percpu偏移量，对于双核CA9，该值为0
        group_offsets[group] = gi->base_offset;
      //该组处理器占用的虚拟地址空间
        group_sizes[group] = gi->nr_units * ai->unit_size;

      //针对组内的每颗处理器
        for (i = 0; i < gi->nr_units; i++) {
            cpu = gi->cpu_map[i];
            if (cpu == NR_CPUS)
                continue;

            …
            unit_map[cpu] = unit + i;
            //计算每颗处理器的percpu虚拟空间偏移量
            unit_off[cpu] = gi->base_offset + i * ai->unit_size;
            …
        }
    }
    pcpu_nr_units = unit;

    …
    //记录下全局参数，留在pcpu_alloc时使用
    pcpu_nr_groups = ai->nr_groups;
    …

    /*
     构建pcpu_slot数组，不同size的chunk挂在不同pcpu_slot项目中
     */
    pcpu_nr_slots = __pcpu_size_to_slot(pcpu_unit_size) + 2;
    pcpu_slot = alloc_bootmem(pcpu_nr_slots * sizeof(pcpu_slot[0]));
  //初始化pcpu_slot数组链头
    for (i = 0; i < pcpu_nr_slots; i++)
        INIT_LIST_HEAD(&pcpu_slot[i]);

    /*
     构建静态 chunk 即 pcpu_reserved_chunk，该区域的物理内存以及虚拟地址都在 int
__init pcpu_embed_first_chunk(…)里分配了
```

```
 */
schunk = alloc_bootmem(pcpu_chunk_struct_size);
…
schunk->immutable = true;
//物理内存已经分配，在这里标志
bitmap_fill(schunk->populated, pcpu_unit_pages);
…
if (ai->reserved_size) {
    //reserved 的空间，在指定 reserved 分配时使用
    schunk->free_size = ai->reserved_size;
    pcpu_reserved_chunk = schunk;
    //定义的静态变量的空间也算进来
    pcpu_reserved_chunk_limit = ai->static_size + ai->reserved_size;
} else {
    schunk->free_size = dyn_size;
    dyn_size = 0;           /* dynamic area covered */
}
schunk->contig_hint = schunk->free_size;

schunk->map[schunk->map_used++] = -ai->static_size;

if (schunk->free_size)
    schunk->map[schunk->map_used++] = schunk->free_size;

/* 动态分配空间，这里构建第一个 chunk，该 chunk 是第一次步进静态变量空间和
reserved 空间使用后剩下的*/
if (dyn_size) {
    dchunk = alloc_bootmem(pcpu_chunk_struct_size);
    INIT_LIST_HEAD(&dchunk->list);
    …
   //记录下来分配的物理页
    bitmap_fill(dchunk->populated, pcpu_unit_pages);

    dchunk->contig_hint = dchunk->free_size = dyn_size;
   /*map 指针更新将静态变量空间和 reserved 空间甩在后面*/
    dchunk->map[dchunk->map_used++] = -pcpu_reserved_chunk_limit;
    dchunk->map[dchunk->map_used++] = dchunk->free_size;
}

/* 把第一个 chunk 链接进对应的 slot 链表，reserved 的空间有自己单独的 chunk:
pcpu_reserved_chunk */
pcpu_first_chunk = dchunk ?: schunk;
pcpu_chunk_relocate(pcpu_first_chunk, -1);

/* we're done */
```

```
    pcpu_base_addr = base_addr;
    return 0;
}
```

1.2.3　Percpu 动态分配内存空间

关于 Percpu 动态分配内存空间有以下基本概念：

（1）每颗处理器的自己 Percpu 动态分配内存空间都有不同的虚拟地址空间，否则同一线程在不同处理器上运行时修改页表页目录项的开销太大。

（2）Chunk 记录每颗处理器一次步进得到虚拟地址空间，对于一颗处理器来说一次步进长度是 ai->unit_size，Percpu 内存的虚拟地址以 Chunk 为基础。

（3）Chunk 中每一次配出的内存用其 map[]指向。

（4）Chunk 根据其 free 的内存长度挂到 Slot 数组的对应链表上。

（5）每次步进仅得到虚拟地址空间，在完成一次分配时才得到对应的物理内存。

本书仅考虑 percpu-vm 的情况。

```
/*动态分配函数 */
static void __percpu *pcpu_alloc(size_t size, size_t align, bool reserved)
{
    static int warn_limit = 10;
    struct pcpu_chunk *chunk;
    const char *err;
    int slot, off, new_alloc;
    unsigned long flags;

    mutex_lock(&pcpu_alloc_mutex);
    spin_lock_irqsave(&pcpu_lock, flags);

    /* 指定 reserved 分配，从 pcpu_reserved_chunk 进行，较简单不讨论*/
    if (reserved && pcpu_reserved_chunk) {
    …
    }

restart:
    /* 根据需要分配内存块的大小索引 slot 数组找到对应链表 */
    for (slot = pcpu_size_to_slot(size); slot < pcpu_nr_slots; slot++) {
        list_for_each_entry(chunk, &pcpu_slot[slot], list) {
         //在该链表中进一步寻找符合尺寸要求的 chunk
            if (size > chunk->contig_hint)
                continue;
         /*chunk 用数组 int*map 记录每次分配的内存块，若该数组用完（该 chunk 仍然还
        有自由空间），则需要增长该 int *map 数组*/
            new_alloc = pcpu_need_to_extend(chunk);
```

```
                if (new_alloc) {
                    spin_unlock_irqrestore(&pcpu_lock, flags);
                    //扩展 int *map 数组
                    if (pcpu_extend_area_map(chunk,
                                    new_alloc) < 0) {
                        …
                    }
                    spin_lock_irqsave(&pcpu_lock, flags);
                    goto restart;
                }
                /*在该 chunk 里分配虚拟内存空间：分割最后一段自由空间，然后重新将该 chunk
                挂到 slot 数组对应链表中*/
                off = pcpu_alloc_area(chunk, size, align);
                //off 大于 0 表示分配成功
                if (off >= 0)
                    goto area_found;
            }
        }

        spin_unlock_irqrestore(&pcpu_lock, flags);
        /*创建一个新的 chunk，这里进行的是虚拟地址空间的分配 */
        chunk = pcpu_create_chunk();
        …
        spin_lock_irqsave(&pcpu_lock, flags);
       //把一个全新的 chunk 挂到 slot 数组对应链表中
        pcpu_chunk_relocate(chunk, -1);
        goto restart;

area_found:
        spin_unlock_irqrestore(&pcpu_lock, flags);

        /* 一次 percpu 内存分配成功，这里要检查该段区域对应物理页是否已经分配，否则将为该
        区域分配对应的物理页并作填充 L1  L2 页表项*/
        if (pcpu_populate_chunk(chunk, off, size)) {
            …
        }

        mutex_unlock(&pcpu_alloc_mutex);

        /* return address relative to base address */
        return __addr_to_pcpu_ptr(chunk->base_addr + off);

        …
}
```

1.3　CpuFreq

简单来说，CpuFreq 由两方面组成，一方面是调频算法，其决定如何调频、何时调频、调频还是 Up 或 Down 处理器；一方面是具体操作由其实现如何调频。其实现为：

（1）governor 监测当前系统的负载情况，即为调频算法。

（2）通过 cpufreq_driver 对 Arm core 外围的时钟电路、供电电路实施操作，达到调频的作用，即为调频操作。

对于大部分 CA9 架构处理器，调频算法都是运行在 CPU0 上，通过 CPU0 来对其他调频、Down 或 Up。

1.3.1　初始化

CpuFreq 初始化的工作主要有以下两个。

在底层将处理器具体操作函数注册到 static struct cpufreq_driver *cpufreq_driver 中，以满足处理器具体调频操作需要。SOC 厂商都必须在 struct cpufreq_driver 里实现自己最底层的调频例程。

```
struct cpufreq_driver {
    …
    int (*init)      (struct cpufreq_policy *policy);
   …
    int (*target)    (struct cpufreq_policy *policy,
                unsigned int target_freq,
                unsigned int relation);
    …
};
```

其中最重要两个例程为：

```
    int (*target) (struct cpufreq_policy *policy, unsigned int
target_freq, unsigned int relation);//该例程实现具体的调频操作
    “int (*init) (struct cpufreq_policy *policy);//该例程初始化 struct
cpufreq_policy
```

在较高级的处理器系统设备抽象层，将 CpuFreq 驱动 static struct sysdev_driver cpufreq_sysdev_driver 注册到处理器系统设备类，以匹配初始化每颗处理器的调频操作。

```
/*
SOC 实现的 BSP 里都要通过调用 int cpufreq_register_driver(struct cpufreq_driver
*driver_data)向 CpuFreq 注册自己的 struct cpufreq_driver，这将产生两个影响：

    将 SOC 相关的 struct cpufreq_driver 暴露给 CpuFreq，governor 将据此进行调频动作
```

```
触发 CPU 设备驱动 struct sysdev_driver cpufreq_sysdev_driver 的注册
这将导致对每一个 CPU 执行 static int cpufreq_add_dev(struct sys_device *sys_dev)
进一步触发 cpufreq_driver->init

 对于 CA9 架构，该函数在具体的底层 BSP 中调用，参数就是 SOC 相关的具体调频操作
 */
int cpufreq_register_driver(struct cpufreq_driver *driver_data)
{
    …
    /*重要静态变量 static struct cpufreq_driver *cpufreq_driver;记录了处理器底
    层相关的调频操作*/
    cpufreq_driver = driver_data;
    /*注册到 struct sysdev_class cpu_sysdev_class，该函数会依次扫描处理器系统设
    备类中的处理器，并为其调用 static struct sysdev_driver
    cpufreq_sysdev_driver 的 static int cpufreq_add_dev(…)函数，从而导致系统
    中调频机制的建立*/
    ret = sysdev_driver_register(&cpu_sysdev_class,
                    &cpufreq_sysdev_driver);
    …
}
```

1.3.2 CpuFreq 策略的建立

对于单颗处理器架构这里很好理解，完成处理器的 struct cpufreq_policy 初始化及 struct cpufreq_governor 的选择即可。但是对于多核处理器来说，这里隐藏着调频算法在哪颗处理器上执行的问题。

```
/*该函数的关键是创建 struct cpufreq_policy，每颗 CPU 初始化、wakeup 的时候都会执行
该函数，但是 struct cpufreq_policy 只在 CPU0 初始化时创建，后续 CPU 在自己的局部数据
per_cpu(cpufreq_cpu_data, cpu);里将索引到该结构*/
static int cpufreq_add_dev(struct sys_device *sys_dev)
{   …
/*如果当前 cpu 被关闭，其不属于调频范畴*/
    if (cpu_is_offline(cpu))
        return 0;
#ifdef CONFIG_SMP
    /*
    对于 CA9 架构，在 CPU0 初始化时创建 struct cpufreq_policy，并且其余处理器的将共
享 CPU0 创建的 struct cpufreq_policy
    其余 CPU 初始化时直接索引到该 struct cpufreq_policy，直接返回
    */
    policy = cpufreq_cpu_get(cpu);
    …
#endif
    /*CPU0 执行到这里，分配 struct cpufreq_policy 结构*/
```

```
    policy = kzalloc(sizeof(struct cpufreq_policy), GFP_KERNEL);
    if (!policy)
        goto nomem_out;
    //分配 struct cpufreq_policy 结构覆盖的处理器范围
    if (!alloc_cpumask_var(&policy->cpus, GFP_KERNEL))
        goto err_free_policy;
…
    //CPU0 找到默认的 GOVERNOR
    if (!found)
        policy->governor = CPUFREQ_DEFAULT_GOVERNOR;

    /* 这里是关键，对于 CA9 架构多核处理器，如 exynos4、imx6q，这里都把 struct
    cpufreq_policy 的 cpumask_var_t cpus 设置为所有的处理器。这里的原因是作为 SMP
    的实现，CA9 每个 CORE 的频率都必须保持一致，一次调频操作对所有的 CORE 都有相同的作
    用。这在之后的 static int cpufreq_add_dev_interface(…)中，所有在线处理器的
    per_cpu(cpufreq_cpu_data, j) 指针都指向了 CPU0 创建的 struct
    cpufreq_policy。而每颗处理器的 per_cpu(cpufreq_policy_cpu, j)都指向了 CPU0
    */
    ret = cpufreq_driver->init(policy);
    …
   //频率的最小最大值
    policy->user_policy.min = policy->min;
    policy->user_policy.max = policy->max;

    …
    ret = cpufreq_add_dev_interface(cpu, policy, sys_dev);
  …
}

static int cpufreq_add_dev_interface(unsigned int cpu,
                    struct cpufreq_policy *policy,
                    struct sys_device *sys_dev)
{

    /*体现 SMP 架构调频特性的操作，所有处理器协同调频步调一致*/
    for_each_cpu(j, policy->cpus) {
        if (!cpu_online(j))
            continue;
        per_cpu(cpufreq_cpu_data, j) = policy;
        per_cpu(cpufreq_policy_cpu, j) = policy->cpu;
    }
    …
    /*这里发送的 CPUFREQ_GOV_START 通知将导致处理器调频算法的启动*/
    ret = __cpufreq_set_policy(policy, &new_policy);
    …
```

```
}
```

1.3.3 Ondemand 调频算法分析

所谓处理器调频算法 struct cpufreq_governor 没有准确的标准，适合的就是最好的，系统中提供了若干调频算法。本文仅选择 struct cpufreq_governor cpufreq_gov_ondemand 分析，这个算法的基本做法是启动一个定时器，定时检查系统负载，然后做出升降频率的决定。

首先 CPUFREQ_GOV_START 消息被 struct cpufreq_governor cpufreq_gov_ondemand 截获：

```
static int cpufreq_governor_dbs(struct cpufreq_policy *policy,
                    unsigned int event)
{
    …
    switch (event) {
    //截获 CPUFREQ_GOV_START 消息
    case CPUFREQ_GOV_START:
…
//定时器初始化
dbs_timer_init(this_dbs_info);
break;
…
}
```

启动定时器来检测系统负载：

```
static inline void dbs_timer_init(struct cpu_dbs_info_s *dbs_info)
{
    …
    /*定时器函数 static void do_dbs_timer(struct work_struct *work)是检测系统
负载的重要机构*/
    INIT_DELAYED_WORK_DEFERRABLE(&dbs_info->work, do_dbs_timer);
    schedule_delayed_work_on(dbs_info->cpu, &dbs_info->work, delay);
}
```

检查系统负载来决定如何调频：

```
static void do_dbs_timer(struct work_struct *work)
{
        …

       /*这里检查系统负载，无非就是累加各个处理器的 idle 时长，与系统设置做比较*/
        dbs_check_cpu(dbs_info);
        …
        //调频
        __cpufreq_driver_target(dbs_info->cur_policy,
```

```
                dbs_info->freq_lo, CPUFREQ_RELATION_H);
            …
    //准备下一次检测
        schedule_delayed_work_on(cpu, &dbs_info->work, delay);
        …
    }
```

在处理器实际实现中，有些 SOC 在实现中通过检查诸如总线忙碌之类的硬件参数以及处理器温度来决定调频操作。而有些系统为了节省成本省掉 PMIC，调频框架并不产生实质的升降频操作。

内核中提供的调频算法一般只作为参考。根据 SOC 实现的不同，几乎大部分厂商的 BSP 都有自己的调频算法。其实现各有不同，但通常情况下有如下共性。

（1）通过调整送入 ARM Core 的 clock 频率及其工作电压完成处理器的频率调整。通常：升频，先调压再调频；降频，先调频再调压。

（2）某些处理器可以保持较高电压的情况下往下调频，但没有一种处理器可以在较低电压下往高调频。

（3）电压的调整针对不同的 PMIC 进行，不同的 PMIC 提供不同的 regulator，在 drivers/regulator 目录下实现不同 PMIC 的 regulator 操作。

（4）调频前后，所有 ONLINE 会接到内核 notify 通知。

（5）对于 SMP 处理器，所有的 CORE 都以相同的频率工作，所以每次调频都是对当前所有 online 的 CORE 进行操作。

涉及调频部分的代码是 SOC 厂商的敏感信息，往往开源代码中并不公开。为避免法律纠纷，这里不再分析具体厂家实现。

1.4　CPU0 bootup CPU1

在 SMP 架构中，尽管每颗处理器运行时所属的地位都一样：一样的调度队列、一样的处理器选择策略。但是在系统引导时，这些处理器的地位是不同的。占有主导作用的是 CPU0，该处理器最先被引导，为其他处理器创建运行环境之后，再 boot up 其他处理器。本节以双核处理器为原型分析，CPU1 指的是非 Booting 处理器。本节分析实例为 Exynos4210。

CPU1 的激活时机有以下情况。

（1）冷启动 CPU0 完成初始化后，叫醒 CPU1。

（2）从 SUSPEND 状态恢复，类似冷启动。

（3）CPUFREQ 监测到当前负载过高。

1.4.1　CPU0 侧策略和动作

在 Bring up CPU1 的过程中，CPU0 主要有两方面的工作。

（1）从 CPU0 跨入 start_kernel 那一刻起，任何一颗 ONLINE 的处理器在任何时刻内必然处在某一进程的 Context 空间内。即使调度队列为空，也必须进入 Idle 进程空间。所以，在 Bring up 其他非 Booting CPU 之前，CPU0 需为其准备 Idle 进程的 Context。

（2）CPU0 需要控制 CPU1 的电源、时钟等硬件开关及时序，引导 CPU1 进入内核入口。

当 CPU0 决定 wakeup CPU1 时，CPU0 执行 cpu_up。

```
//CPU0 的执行路线
int __cpuinit __cpu_up(unsigned int cpu)
{
    …
    // Idle 指向待唤醒处理器调度队列中的线程
    if (!idle) {
    …
    } else {
        …
        //为即将唤醒的处理器准备一个 idle 线程
        init_idle(idle, cpu);
    }

    //把 CPU1 要用到的页表页目录准备好
    pgd = pgd_alloc(&init_mm);
    pmd = pmd_offset(pgd + pgd_index(PHYS_OFFSET), PHYS_OFFSET);
    *pmd = __pmd((PHYS_OFFSET & PGDIR_MASK) |
             PMD_TYPE_SECT | PMD_SECT_AP_WRITE);
    flush_pmd_entry(pmd);
    outer_clean_range(__pa(pmd), __pa(pmd + 1));

    //把为 CPU1 准备启动的参数放在 secondary_data 里
    secondary_data.stack = task_stack_page(idle) + THREAD_START_SP;
    secondary_data.pgdir = virt_to_phys(pgd);
    __cpuc_flush_dcache_area(&secondary_data, sizeof(secondary_data));
    outer_clean_range(__pa(&secondary_data), __pa(&secondary_data + 1));


    //叫醒 CPU1
    ret = boot_secondary(cpu, idle);
    …

    return ret;
}

//CPU0 唤醒 CPU1 的硬件操作
int __cpuinit boot_secondary(unsigned int cpu, struct task_struct *idle)
{
```

```
    //CPU0 把要叫醒的 CPU 写在 pen_release 中
    pen_release = cpu;
    __cpuc_flush_dcache_area((void *)&pen_release, sizeof(pen_release));
    outer_clean_range(__pa(&pen_release), __pa(&pen_release + 1));

    /*CPU0 把 CPU1 的电源域打开，这时在另一侧，CPU1 开始启动，CPU1 跑到 boot monitor
中。CPU1 将在 boot monitor 等待*/
    if (!(__raw_readl(S5P_ARM_CORE1_STATUS) & S5P_CORE_LOCAL_PWR_EN)) {
        __raw_writel(S5P_CORE_LOCAL_PWR_EN,
                S5P_ARM_CORE1_CONFIGURATION);
        timeout = 10;

        /* 等待检查 CPU1 是否完成硬件 Reset，并跑到 Bootrom 里相应位置*/
        while ((__raw_readl(S5P_ARM_CORE1_STATUS)
            & S5P_CORE_LOCAL_PWR_EN) != S5P_CORE_LOCAL_PWR_EN) {
            if (timeout-- == 0)
                break;

            mdelay(1);
        }

        …
    }
    …

    /* CPU0 把 CPU1 启动跳转地址告诉寄存器 CPU1_BOOT_REG，这时 CPU1 还没有打开 MMU，
    放置的是物理地址，CPU1 侧将查看寄存器 CPU1_BOOT_REG。CPU1 不走 BL1，CPU0 的休眠
    唤醒才走 BL1*/
        if (!__raw_readl(CPU1_BOOT_REG)) {
            //CPU1 出了 Bootrom 后的地址
            __raw_writel(BSYM(virt_to_phys(s5pv310_secondary_startup)),
                    CPU1_BOOT_REG);
            smp_cross_call(cpumask_of(cpu));
        }
        //CPU1 重新设置了 pen_release，CPU0 这边的控制工作到此为止
        if (pen_release == -1)
            break;

        udelay(10);
    }
    …
}
```

1.4.2　CPU1 侧执行路线

CPU1 通常由自己电源域单独供电。当 CPU1 的电源域被打开后，CPU1 直接以 CPU0

的当前频率起跳，然后进入处理器的片上 bootrom 里，在一个与 CPU0 约定的寄存器里查找自己的启动地址后，再从这个约定地址直接进入内核。对于 CA9，这个进入的地址通常被命名为函数：xxx_secondary_startup。

```
//非 Booting 处理器入口地址
ENTRY(s5pv310_secondary_startup)
    /*读取本 CPU 的 ID，参见 DDI0388F_cortex_a9_r2p2_trm.pdf*/
    Mrc p15, 0, r0, c0, c0, 5
    And r0, r0, #15
    Adr r4, 1f
    Ldmia r4, {r5, r6}
    Sub r4, r4, r5
    Add r6, r6, r4
    /*查看 pen_release 的 CPU  ID 是否是自己的 ID*/
pen:ldr r7, [r6]
    cmp r7, r0
    bne pen
    /* pen_release 的 CPU  ID 是自己的 ID，__cpu_up()在叫自己*/
    b   secondary_startup

1:  .long   .
    .long   pen_release

// arch/arm/kernel/head.S

ENTRY(secondary_startup)
    …
    adr r4, __secondary_data
    /*r12 被放入地址__secondary_switched*/
    ldmia   r4, {r5, r7, r12}       @ address to jump to after
    sub r4, r4, r5                  @ mmu has been enabled
    ldr r4, [r7, r4]                @ get secondary_data.pgdir
    /*执行完 proc init 函数之后返回到__enable_mmu 执行*/
    adr lr, BSYM(__enable_mmu)      @ return address
    /*__enable_mmu 执行之后会跳到 r13 执行。这里 r13 被放入__secondary_switched
的地址*/
    mov r13, r12              @ __secondary_switched address
     ARM(   add pc, r10, #PROCINFO_INITFUNC ) @ initialise processor
                             @ (return control reg)
    THUMB(  add r12, r10, #PROCINFO_INITFUNC    )
    THUMB(  mov pc, r12                 )
ENDPROC(secondary_startup)

ENTRY(__secondary_switched)
    Ldr  sp, [r7, #4]              @ get secondary_data.stack
```

```
    mov fp, #0
    //最终执行 secondary_start_kernel
    b   secondary_start_kernel
ENDPROC(__secondary_switched)

    //__secondary_data 处存放的内容
    .type   __secondary_data, %object
__secondary_data:
    .long   .
    .long   secondary_data
    .long   __secondary_switched
```

接下来讨论一个特殊情况，即关闭 CPU1 失败后的再引导。这种情况发生在试图关闭 CPU1，但是底层关闭操作动作失败。

```
//CPU1 关闭的最后一步
void __ref cpu_die(void)
{
    …
    //关掉 CPU1，最底层的 CPU1 关闭操作
    platform_cpu_die(cpu);
    /* 关闭失败，但是这时又不能直接返回 Idle，需要重新走一次引导流程。正常情况下 CPU1
    的 wakeup 不走这条路，只有在 Die 失败时才走这里*/
    __asm__("mov    sp, %0\n"
    "   b   secondary_start_kernel"
        :
        : "r" (task_stack_page(current) + THREAD_SIZE - 8));
}
```

1.5　CPU1 的关闭

1.5.1　关闭时机

在运行中系统根据负载或者电源策略而改变，为节省电力，在必要时关闭非 Booting 处理器，而作为系统中坚持到最后的 Booting 处理器，在系统 Suspend 的时候也会关闭。本节仍以双核处理器为分析原型，分析非 Booting 处理器即 CPU1 的关闭时机。

CPU1 关闭时机有以下两个情况。

（1）在 CpuFreq 机构监测到当前负载过低时，在某些平台实现时会关闭该处理器。内核负载检测机构会不停计算系统负载，在负载降低到一定阈值之后，将关闭 CPU1。这里值得关注两个问题，首先，在某些 SOC 实现中硬件能够通过监控处理器状态、总线繁忙程度来判断负载情况，这时对系统负载的检测就不需要通过内核定时计算负载来完成；再者，事实上在负载变化时是先调频还是先关闭、打开处理器，每种 SCO 厂商采用的策略都不一

样。对于核心数目较多的处理器笔者认为开关优先是较好的策略，对于双核处理器调频优先更为合理。

```
//定时监测系统负载
static void dbs_check_cpu(struct cpu_dbs_info_s *this_dbs_info)
{
…
//平均负载是否小于阈值
    if (avg_load < dbs_tuners_ins.down_threshold) {
        if (policy->cur == policy->min) {
            /* 当前活跃处理器较多，且 hotplug_out_avg_load 小于阈值*/
            if (num_online_cpus() > 1 && hotplug_out_avg_load <
                    dbs_tuners_ins.down_threshold) {
                cpu_down(1);
            }
        …
        }
…
```

（2）在进入 Suspend 时，操作系统电源决策机构如 Android 的 PowerManagement 机制或者用户自身，决定要将系统休眠时，CPU1 将首先被关闭。

```
//进入休眠，该函数的调用来自操作系统层的触发
static int suspend_enter(suspend_state_t state)
{    …
    //关闭非 booting 处理器
    error = disable_nonboot_cpus();
    …
}
```

1.5.2　CPU1 关闭操作

CPU1 的关闭由两方面组成：一是决策机构；二是 CPU1 本身。其关闭过程也分为以下阶段。

（1）决策处理器上运行的决策算法决定关闭 CPU1，CPU_DOWN_PREPARE 类型通知被发出，导致 cpu_active_bits 的 CPU1 对应位被清除。

（2）将 static int __ref take_cpu_down(void *_param)操作挂到 CPU1 的 struct cpu_stopper 中，并等待 CPU1 完成操作。

（3）CPU1 上执行 static int __ref take_cpu_down(void *_param)，将 cpu_online_bits 上的 CPU1 对应位清除；将 CPU1 上中断转移走，并关闭局部时钟中断；更关键的是将系统中所有线程允许运行位图上的 CPU1 位清除；发出 CPU_DYING 通知，导致 CPU1 的 wake_list 链表上线程被加入 CPU1 运行队列（这时其运行位图已经去除了 CPU1 对应位），接下来执行 static void migrate_tasks(unsigned int dead_cpu)，将 CPU1 上线程迁移到别的处理器上。

（4）CPU1 继续运行，CPU1 上将只剩 IDLE 线程可以运行。决策处理器继续往下运行等待 CPU1 上调度上 idle 线程。

（5）CPU1 进入 idle task，在 CPU1 侧会清除 CACHE 之类的操作，然后在大部分 CA9 的实现中 CPU1 都会写入与决策处理器约定的某个寄存器，然后执行 WFI。

（6）决策处理器接下来与 CPU1 约定的寄存器被置位后，关闭 CPU1 电源。

接下来分析上述动作的源代码实现，首先看每个 CPU 的 struct cpu_stopper 结构。它用来处理该 CPU 被关掉之前的一些工作。

```
struct cpu_stopper {
    spinlock_t        lock;
    struct list_head    works;        /* list of pending works */
    struct task_struct  *thread;     /* stopper thread */
    bool               enabled;       /* is this stopper enabled? */
};
```

struct cpu_stopper 结构的成员变量 struct task_struct*thread;执行的函数为：

```
static int cpu_stopper_thread(void *data)
{
不断从 struct list_head  works;取出 work 执行
}
```

在被关掉的 CPU1 的 cpu_stopper 线程里执行的函数：

```
static int __ref take_cpu_down(void *_param)
{
    struct take_cpu_down_param *param = _param;
    …
    //把这个 CPU online 位清零，并关掉这颗 CPU 的中断和时钟
    err = __cpu_disable();
    …
   //发出 CPU_DYING 通知
    cpu_notify(CPU_DYING | param->mod, param->hcpu);

    //把该 CPU 运行队列的 task 迁移走
    if (task_cpu(param->caller) == cpu)
        move_task_off_dead_cpu(cpu, param->caller);
     …
    //把这个 CPU 的 idle 线程调度起来
    sched_idle_next();
    return 0;
}


//在被关掉的 CPU 的 idle 函数中
void cpu_idle(void)
```

```
{
…
#ifdef CONFIG_HOTPLUG_CPU
//这颗 CPU 对应的 online 位已被清零，所以这颗 CPU 进入 cpu_die()
            if (cpu_is_offline(smp_processor_id()))
                cpu_die();
#endif
…
}
```

1.6 ARM 处理器展望

1.6.1 ARM 架构处理器的演进

ARM 公司不断推进 ARM 架构处理器的演进，在不同时期有着不同代表作。从彪炳史册的 ARM7/9，到横空出世的 Cortex A8/9，再到尚处研发状态的 64 位 V8 架构处理器。每一代的 ARM 处理器都有着自己的历史使命，ARM 7/9 的任务是开疆拓土，实现了 ARM 的普及，Cortex A8/9 完成了支撑智能手机推翻功能机的革命，64 位架构目标是进军服务器领域。

然而这些都不是本节的重点，本节要分析的是 Cortex A7/A15 架构处理器。此处理器似乎仅仅是 CA8/9 的补充，没有明确的目标。但是 Cortex A7/A15 身上却蕴藏至今尚未被真正释放的能力——实现系统软件架构的革命。

1. Cortex A15

Cortex A15 主要有以下 3 个方面的改进。

（1）硬件支持虚拟化

ARM 在 CA15 之后发布的处理器都支持硬件虚拟化，CA15 扩展出了一个更高优先级的模式，将作为 hypervisor 的 kernel 和 user mode 都运行在这个模式，这种模式下提供了 PL2 的 IPA 到 PA 的转换以及 VGIC。这样可以由 Guest OS 内核自主处理自己管辖范围内的页异常，而不必 hypervisor 大量的介入。笔者认为这是处理器虚拟化最关键的地方，不然大量的页异常掉到 hypervisor，不仅性能受到影响，而且架构上也是畸形的。

（2）40 位物理地址

32 位物理地址的问题以前在 X86 体系下出现过，现在又开始困扰 ARM 系统了。感谢进程间虚拟地址隔离，4G 的虚拟地址仍能满足进程的需求，但是 4GB 的物理内存很快达到手机、平板的极限。40 位物理地址为此而生，这样可以把任意一个 4G 的虚拟地址空间甩到 1024G 的物理内存的任何地方。

（3）流水线的增加和 Neon 流水线的融合

Cortex A15 像其他时代的 ARM 演进一样，都会扩充流水线的，改进分支预测等，但是值得一提的是 NEON 不再放在整数流水线的后面，当译码出 NEON 指令后再甩给 NEON。

A15 里的 NEON 跟其他的流水线同时接受调度。

从 Cortex A15 的设计可看到 PC 级处理器的身影，更复杂的逻辑实现带来更高的性能。ARM 一直以低功耗著称，而 Intel 的处理器则反之。是 Intel 的技术不如 ARM？是 Intel 的工厂比不过 Tsmc、SAMSUNG？显然都不是。原因在于：对高性能的追求，必然要用到更先进的处理器设计，更复杂的逻辑，导致在同样工艺水平下 Intel 处理器占用更大的芯片面积，自然功耗就上来了。当 Intel 忍痛阉割掉 Medfiled 里面的那些先进的复杂部件，自然功耗降下来了。这似乎是个围城，为了追求高性能， A15 充分体现了 ARM 工程师的智慧，然而导致了芯片复杂度的升高，从而给 ARM 带来了困扰 Intel 的噩梦——功耗。

而反观 Intel，由于其拥有独步天下的半导体工厂，在逐渐改进工艺的水平的情况下，可以使用更多复杂运算单元而保持功耗不变或下降。而 ARM 就玩不转了，因为 ARM 的投片工厂可能是台积电、三星、GlobalFoundries 等大厂，但也有可能是一些工艺水平落后若干代的小厂。所以 ARM 的逻辑设计必须考虑到这些工厂的实现能力，一些先进复杂运算单元是无法及时在其处理器设计中使用的。所以一些大的半导体公司，干脆直接重新设计其 ARM 处理器，然后在使用台积电等大厂最先进的工艺来生产，这似乎起到了些较好效果比如 Qualcomm 的 Snapdragon。但是笔者以为，这还是比不了人剑合一的 Intel。

但是，天下是天下人的天下。ARM 阵营有着高效、低成本的特点，不仅横行移动市场而且大有谋取 PC、Server 的趋势。

2. Cortex A7 与 Big.Little

Cortex A7 与 CA15 共享指令集，但是 CA7 不能乱序执行，性能比 CA8 略低，CA7 似乎是 CA15 的缩小版，其 Die Size 只有 A15 的 1/4，ARM 发布这个架构处理器的目的似乎是直奔功耗而来。除了可以作为一颗单独的处理器，CA7 还可以与 CA15 搭配工作，构成 big.little 架构，这个架构的关键如下。

（1）大部分系统运行的时候都不需要太多处理器能力，一个多核系统的大部分时间都是将其他的 Core 关掉的。这时 Cortex A7 就能满足要求，可以把耗电的 A15 关闭掉。但系统遇到计算压力的时候再 bring up A15，以实现高性能。

（2）由于 A15 与 A7 是软件完全兼容的，所以可以套用内核现有 SMP 支持架构而不需要大规模改动。

如果指令集相同，在不修改内核模型的情况下，不只是 A15 搭 A7，A9 搭 A5 也是一样可行的。

由于其小巧、省电且保持相当计算能力的特点。A7 已成为低功耗及价格敏感市场的新兴力量，MTK 以及几乎所有的大陆手机处理器厂商在近几年的主力产品都使用多核 A7。而一些老牌嵌入式厂商也将 A7 作为计算中能力要求不高、且价格敏感市场的主力。

1.6.2　TrustZone

TrustZone 是 ARM 架构的安全扩展，在 ARM11 时首先出现，其主要思想是通过将处理器扩展出一个新的 Security 特权状态以完成安全性相关工作。但是尽管安全领域的市场需求强烈而迫切，TrustZone 自问世以来一直未被业界广泛接纳。其原因在于：

（1）TrustZone 要求在 Security 特权状态构建一个全新的包括内核、应用在内的生态系统，而这个生态系统一直没有成长起来。

（2）由于 TrustZone 本身的架构问题，导致在其上实现软件体系会遭遇如下问题。

① 首先，若所有的处理器核心全进入 Security 特权状态，势必修改当前内核模型，Non-Security 状态下平滑运行的状态被打断，而所有的 Core 都进入 Security 特权状态又无必要，Security 特权状态下的主要应用只有加解密算法和输入。若只有一个处理器进入 Security 特权状态，架构体系上又不够完美。

② 即使不考虑架构设计和运行效率，Security 特权状态另外一个麻烦在于，当在 Security 特权状态的某个 Core 操作外设时，势必与 Non-Security 状态下的系统软件冲突，这与虚拟化中 Guest 访问非虚拟设备的问题一样，后面再详细介绍。

③ 另外，TrustZone 机制的一个基础在于总线访问时外设依赖其当前 AXI 交易的安全位来做出如何响应，但是往往 Security 特权状态需要控制的外设 IP 设计并没有这个考虑。

所以，尽管 TrustZone 有着强烈而广泛的市场需求，但是由于其本身对现有软件体系发起的挑战，使得业界不得不另外寻找方式以解决安全问题。

其实从某种角度来看，TrustZone 的 Security 与 NON-Security 模式更适合虚拟化架构，在安全模式可以使用独立的页表页目录、而且安全模式下也有对应于非安全模式下的 7 种模式，类似于 Intel 的处理器虚拟化架构的 Root 和 Guest 的关系。这样 KVM 运行在安全模式下还能保持内核-SVC 与用户-USR 的架构，非常完美。

兼容是处理器演进的第一规律，所有挑战这个规律的处理器都是找死，即使不成功的处理器设计也得兼容。为安全架构另外再设计一个模式，然后稍加修改 TrustZone 以实现 ARM 架构完美虚拟化。但是 ARM 没有这样做，而是选择将 HYP 塞在 PL1 的下面，这是一个尚可接受但并不完美的虚拟化架构，也许是这个规律在起作用。注意，笔者是说“也许”。

1.6.3 ARM Virtualization

ARM 架构在 CA15 以后发布的 A 系列处理器都支持虚拟化扩展。在这种体系下实现虚拟化有多种方案可以选择，尽管所有的方案理论上都可行，做 DEMO 工程没有问题，但是如果能够产品化，笔者认为只有 KVM 更适合 ARM 虚拟化体系，其原因如下。

（1）在一个没有虚拟化 IO 支持的硬件架构上，由于坚持使用独立的 Hypervisor，嵌入式虚拟化的实现不得不允许 Guest OS 直接访问硬件设备。由此带来诸多隐患：多个 Guest OS 和 Hypervisor 陷入硬件访问冲突、中断分发、DMA 控制的深渊。因为控制外设不是仅仅读写，还要考虑到其休眠唤醒、中断分发等情况。比如 Guest OS 自身的休眠对设备的影响，设备的休眠与电源域相互影响，而电源域包括多个设备必然在 HyperVisor 里控制，这自然导致与其他 Guest OS 相互影响。可见由于需要和硬件平台过度耦合，难以扩展，嵌入式虚拟化只适合硬件复杂度较小的硬件平台，或者只使用部分功能的复杂硬件平台。

（2）而诸如 Xen 之类虚拟化方案修改 Linux BSP 成为 Domain0，这样所有硬件访问都可以被 Domain0 接管，理论上是可行的。但是笔者认为这不是一个适合 ARM 虚拟化体系的好架构。而且 Xen 架构设计之初是为了适配没有硬件支持 X86 处理器，在有着硬件支持

的处理器上这个虚拟化架构过于臃肿，使得简单问题复杂化。

（3）另外，抛开架构不谈，将 Linux BSP 改成 Domain0 的工程量也不可同日而语。在 CA8 以后， ARM SOC 体系已经不再像 ARM9 那么简单了。SOC 构成普遍出现了 GPU、VPU 等多个主设备，层层交叉互联结构的总线，独立而又相互依赖的电源域、时钟域，层级的中断控制器与多核情况下的中断分发。要完美实现对所有硬件组件的控制势必有很多验证、测试工程量。

（4）更进一步，还存在一个非技术因素，在于 ARM SOC 界的游戏规则发生改变，半导体厂商不再像以前一样公开 SOC 中所有硬件信息。很多硬件 IP，厂商虽然开源了 BSP，但是却没有任何资料描述，而诸如 GPU 之类的驱动只以二进制的形式发布。这使得 Linux BSP 改成 Domain0 成为非原厂而不能为的事情。

所以，笔者认为最适合 ARM 的虚拟化架构是 KVM，其特点在于能够遵循现有以 Linux BSP 管理硬件的游戏规则；避免 Guest OS 直接访问外设以提高移植性；可以采用 Frontend+Backend 驱动模型，以提高其性能。

ARM KVM 的实现完全符合 KVM 原有架构，只是在处理器相关部分与 ARM 虚拟化扩展做了适配。其实现如下：

（1）ARM 虚拟化扩展出的 Hyp 模式拥有最高优先权 PL2，但是这个模式下仅作为 HOST 与 Guest 切换的中转。

（2）HOST Linux 工作在 PL0 和 PL1 状态下，其 Context 被保存在 Hyp 的 stack 中。

（3）Guest Linux 工作在 PL0 和 PL1 状态下，其 Context 被保存在其虚拟处理器相关结构中。

Hyp 模式下一张__kvm_hyp_vector 异常表是 ARM KVM 架构适配的核心，搭起 Host 与 Guest 之间的桥梁。

```
__kvm_hyp_vector:
    .globl __kvm_hyp_vector
    W(b)    hyp_reset
    W(b)    hyp_undef
    W(b)    hyp_svc
    W(b)    hyp_pabt
    W(b)    hyp_dabt
    W(b)    hyp_hvc
    W(b)    hyp_irq
    W(b)    hyp_fiq
```

根据 ARM KVM 的设计，hyp_undef、hyp_svc、hyp_pabt、hyp_dabt 四种异常目前被认为是不可能出现在 Hyp 模式的，这几种异常向量都被定义为没有意义的 bad_exception。

1. HVC 异常

作为两侧 HVC 指令异常的入口，hyp_hvc 异常向量是 Hyp 模式下最重要的异常向量：

```
hyp_hvc:
    /*HVC 调用通常携带参数而来，这里先将其保存*/
```

```
    push    {r0, r1, r2}
    /*HSR 寄存器里记录了 HVC 异常的产生是否是由于 PL1 发生 HVC 异常调用，这里读取 HSR
    寄存器，并检查是否是 PL1 遇到了 HVC 指令*/
    mrc p15, 4, r1, c5, c2, 0   @ HSR
    lsr r0, r1, #HSR_EC_SHIFT
    …
    cmp r0, #HSR_EC_HVC
    bne guest_trap      @ Not HVC instr.

    /*
     确认是 PL1 遇到 HVC 指令，接下来要通过检查 VMID 区分是来自 Host 还是 Guest 的调用
     */
    mrrc    p15, 6, r0, r2, c2
    lsr     r2, r2, #16
    and     r2, r2, #0xff
    cmp     r2, #0
   /*来自 Guest 的 HVC 调用将跳转至 guest_trap*/
    bne guest_trap      @ Guest called HVC
/*来自 HOST 的 HVC 调用走到这里*/
host_switch_to_hyp:
    /*弹出刚才保存的 HOST 送下来参数*/
    pop {r0, r1, r2}
    /*保存 SPSR*/
    push    {lr}
    mrs lr, SPSR
    push    {lr}
    /*HOST 调用 kvm_call_hyp，将 Hyp 里的调用地址放在 r0 里，struct kvm_vcpu 指针
    放在 r1 里，这里把跳转地址放入 lr，struct kvm_vcpu 指针放入 r0。这里还要处理好 r2、
    r3，以备将来之需*/
    mov lr, r0
    mov r0, r1
    mov r1, r2
    mov r2, r3
    …
    /*跳转到__kvm_vcpu_run 准备恢复 Guest*/
    blx lr          @ Call the HYP function
    /*从 Guest 返回到 Host 的最后步骤，将从这里弹到 PL1*/
    pop {lr}
    msr SPSR_csxf, lr
    pop {lr}
    /*Hyp 模式遇到 eret，弹到 PL1*/
    eret

/*Guest hvc 异常走到这里*/
guest_trap:
```

```
    //r0 存放 VCPU 的 Contex 的指针
    load_vcpu           @ Load VCPU pointer to r0
    str r1, [vcpu, #VCPU_HSR]

    /*若从 Guest 而来，除了 HVC 调用，还可能是 MMU 转换异常，如对设备寄存器地址的访问*/
    lsr r1, r1, #HSR_EC_SHIFT
    cmp r1, #HSR_EC_IABT
    mrceq   p15, 4, r2, c6, c0, 2   @ HIFAR
    beq 2f
    cmp r1, #HSR_EC_DABT
    /*不是转换问题，直接走返回路径*/
    bne 1f
    /*接下来读取 Cp15 获取必要信息，在这里展开，继续分析往 Host 的跳转*/
    …
    /*记录下是 HVC 调用*/
1:  mov r1, #ARM_EXCEPTION_HVC
    /*Host 的跳转函数*/
    b   __kvm_vcpu_return

   /*到这里，说明 Hyp 对 Guest 产生的异常感到绝望，最有可能是 Guest 内核里的野指针，交
   给 Guest 自己了断*/
4:  pop {r0, r1}        @ Failed translation, return to guest
    mcrr    p15, 0, r0, r1, c7  @ PAR
    clrex
    pop {r0, r1, r2}
    eret
```

接下来分析 Hyp 模式中对 Host 向 Guest 切换的支持。

```
/*这个函数脉络很清晰，即把 Host Context 压栈，恢复 Vcpu，eret*/
ENTRY(__kvm_vcpu_run)
    …
    /*这是一个宏把 Host Context 压栈，值得注意的是这里的 Context，除了通用寄存器、状
    态寄存器，还有 CP15 里的相关信息*/
    save_host_regs
    …
    /*恢复 vcpu 状态*/
    restore_guest_regs
    clrex
    /*弹到 Guest*/
    eret
```

从 Guest 返回 Host 与从 Host 恢复 Guest 类似，只是弹向 Host 的操作不是在这里进行的，参见上文。

```
/*Guest 向 Host 返回，来至 Guest 的 Hvc 异常*/
__kvm_vcpu_return:
```

```
    /*保存 Guest 的 Context*/
    save_guest_regs
    …
   /*恢复 Host 的 Context，这里的关键是栈里的 lr 并不是 Host 的 lr，而是 Host 切换到
   Guest 之前的下一条地址，因为当时 Hyp 使用了 blx，而不是向 Guest 向 Host 时使用的 b。
   其实无论 host、hyp、guest 本身就是一个线程，至此弹掉 vcpu，接着之前的栈轨迹，非常符合
   逻辑*/
    restore_host_regs
    …
    /*返回到__kvm_vcpu_run 之前的下一条指令 */
    bx  lr          @ return to IOCTL
```

2. 中断与快速中断异常

接下来再分析 hyp 异常表的另外两个异常处理。

```
/*中断异常向量函数*/
hyp_irq:
    push    {r0, r1, r2}
    /*保存下原因，中断导致的 Guest Exit */
    mov r1, #ARM_EXCEPTION_IRQ
    load_vcpu             @ Load VCPU pointer to r0
    /*继续向 Host 返回，与 hvc 异常操作一样*/
    b   __kvm_vcpu_return
    .align
/*快速中断异常处理，与中断异常处理无异*/
hyp_fiq:
    b   hyp_fiq
```

针对 ARM KVM 进行再深层次分析。Host Linux 并没有工作在 Hyp 模式下，而逻辑上 Hyp 模式就好比一扇连接 HOST 世界与 GUSET 世界的大门，与 ARM 处理 SE 与 NON-SE 状态切换的手法如出一辙，都是通过异常来实现。尽管 HOST 与 Guest 都工作在 PL0 和 PL1，但是 HOST 是有着优先级特权的，正是这扇大门将 ROOT 世界和 Guest 世界隔离开来。

Hyp 模式下的所有工作都可以通过处理器硬件扩展来实现，这样处理器模式就可以精简掉 Hyp 模式，但是 ARM 选择了这种轻量级方式，ARM 的生存哲学又一次体现在其演进道路上。

第 2 章　异　　常

异常，这个词非常形象地描述了诸如中断、MMU 转换、系统调用等需要内核介入的情况与系统运行时的关系。系统的绝大部分工作时间应该是属于应用程序的，应用程序运行才是系统工作的正常情况，内核介入的时候都属于不正常状态。

操作系统设计的目的是为了运行应用，无论内核再强大，它在系统中都是应该位居幕后的、默默无闻的，应该像大地一样稳稳的支撑，而不用过多的关注。若一个内核需要时不时在系统中展露自己，那一定不是一个好的内核。

ARM 处理器设计了若干种模式以便内核处理对应的情况，本章分析 ARM Linux 对异常处理的基本框架，并着重分析与虚拟内存管理密切相关的数据异常以及影响整个内核机理的中断异常。

传统 ARM 体系的 7 种异常类型如下。

（1）复位异常。即复位，从 reset 异常向量重新执行。

（2）数据异常。即访问数据时由于数据页面不存在或者页表项没有建立等原因引起的异常。

（3）预取指令异常。即访问处理器取指令时由于代码页面不存在或者页表项没有建立等原因引起的异常。

（4）快速中断请求异常。快速中断 Fiq。

（5）中断请求。中断 IRQ。

（6）软件中断异常。用于系统调用，即 SWI，内核维护一张系统调用表，一旦 SWI 陷入内核，内核根据系统调用号索引到相应 API，并依据用户态送下来的参数调用相应函数 API，完成后走类似用户态中断、数据及取指异常的路径返回用户态。

（7）未定义异常。

2.1　异常向量表

2.1.1　异常进入

根据 ARM 处理规范，ARM 的异常向量表可以位于 0X0000 或者 0XFFFF0000，这两个位置处理器可以任选实现。现代处理器一般都是可以通过软件来动态选择异常向量表的位置。又鉴于大部分 SOC 实现都是使用 0XFFFF0000，因此本书认为 0XFFFF0000 就是 ARM 异常向量表地址。

在 Linux 内核中，ARM Linux 的异常向量表位于 arch/arm/kernel/entry-armv.S 中。其地

址范围为__vectors_start 到__vectors_end，依次为复位异常、未定义指令异常、软件中断异常、取指令异常、取数据异常、保留异常、中断异常、快速中断异常。

```
    .globl  __vectors_start
__vectors_start:
    swi SYS_ERROR0
    b   vector_und + stubs_offset
    ldr pc, .LCvswi + stubs_offset
    b   vector_pabt + stubs_offset
    b   vector_dabt + stubs_offset
    b   vector_addrexcptn + stubs_offset
    b   vector_irq + stubs_offset
    b   vector_fiq + stubs_offset

    .globl  __vectors_end
```

其中每种类型的异常都会跳转到一个异常处理代码段。尽管异常种类很多，但是这些异常代码段的实现是如下形式：

```
vector_stub XXX, XXX_MODE, 4  //调用宏 vector_stub，其中 XXX 为某种异常
/*以下又是一个跳转表，该跳转表描述了处理器在某种模式下发生该种异常的处理方式，其中最重要的是 usr 和 svc 模式，因为 Linux 系统下只用到这两个模式。而 ARM 只有 7 种模式（不考虑 CA15 的 Hypervisor 模式，也不考虑 Trustzone 的 Monitor 模式），但是该跳转表确定了 16 项，其原因在于状态寄存器的 0～4 位都被用来标记处理器模式了，所以为了防止某个异常的出现而搞的内核不知所措，该跳转表被定义为 16 项*/
    .long   __XXX_usr           @ 0  (USR_26 / USR_32)
    .long   __XXX_invalid       @ 1  (FIQ_26 / FIQ_32)
    .long   __XXX_invalid       @ 2  (IRQ_26 / IRQ_32)
    .long   __XXX_svc           @ 3  (SVC_26 / SVC_32)
    .long   __XXX_invalid       @ 4
    …
    .long   __XXX_invalid       @ e
    .long   __XXX_invalid       @ f
```

异常处理宏的实现如下：

```
    .macro  vector_stub, name, mode, correction=0
    .align  5

vector_\name:
    .if \correction
    sub lr, lr, #\correction
    .endif

    /*发生异常时 arm 会把 cpsr 保存在 spsr 里，所以检查 spsr 就能查出，异常发生前处理器的模式*/
    @ Save r0, lr_<exception> (parent PC) and spsr_<exception>
```

```
    @ (parent CPSR)
    stmia   sp, {r0, lr}      @ save r0, lr //lr 存放异常发生前 pc 的值，r0 是 swi
                                            的参数
    mrs lr, spsr              //取出 spsr 的内容到 lr 中
    str lr, [sp, #8]          @ save spsr

    /*不管是从什么模式进入异常的，在 Linux 内核使用 SVC 模式，所以将处理器的状态切换到
    SVC 模式*/

    @ Prepare for SVC32 mode.  IRQs remain disabled.
    mrs r0, cpsr  //取出 cpsr 的内容
    eor r0, r0, #(\mode ^ SVC_MODE) //置为 SVC
    msr spsr_cxsf, r0 //写回 cpsr

    @
    @ the branch table must immediately follow this code
    @
    and lr, lr, #0x0f  //取出异常发生前 cpsr 的模式状态位
    mov r0, sp
    /*根据模式确定跳转值。当前 pc 值作为基址，pc 的值为下一条指令地址，所以跳转的地址应
    该是紧跟当前这段代码的一个地址数组，即上问提到 16 个跳转项*/
    ldr lr, [pc, lr, lsl #2]
    movs    pc, lr            @ branch to handler in SVC mode
ENDPROC(vector_\name)
    .endm
```

2.1.2　异常表的构建

以上定义了内核的异常向量表和对应的处理函数，编译后被链接到内核 Image，但是内核 Image 被加载到物理内存的地址不是使得该向量表正好能被映射到虚拟地址 0xffff0000，所以内核还需要将异常向量表搬到 0xffff0000，而向量表与对应的处理函数的跳转使用指令 B，为相对跳转指令，指令 B 一共才 32 位长，除去 B 的编码，留给携带的偏移量只有 32M，所以还得把处理函数处理到 0xffff0000 附近，且偏移量要跟链接里的相等。

1. 位于 arch/arm/kernel/traps.c

```
void __init early_trap_init(void)
{
//SOC 使用的异常向量表基地址可认为 CONFIG_VECTORS_BASE=0XFFFF0000
    unsigned long vectors = CONFIG_VECTORS_BASE;
    extern char __stubs_start[], __stubs_end[];
    extern char __vectors_start[], __vectors_end[];
    extern char __kuser_helper_start[], __kuser_helper_end[];
    int kuser_sz = __kuser_helper_end - __kuser_helper_start;
```

```
    /*
     * Copy the vectors, stubs and kuser helpers (in entry-armv.S)
     * into the vector page, mapped at 0xffff0000, and ensure these
     * are visible to the instruction stream.
     */

//把异常向量表移动到 0XFFFF0000，__vectors_end - __vectors_start 为异常向量表长度
    memcpy((void *)vectors, __vectors_start, __vectors_end - __vectors_start);

/*把这些 vector_xxx 函数及.LCvswi 移动到 0XFFFF0000+0x200 的位置，__stubs_start 为 vector_xxx 及.LCvswi 函数开始位置，__stubs_end 为结束位置*/
    memcpy((void *)vectors + 0x200, __stubs_start, __stubs_end - __stubs_start);

    …
    //有代码移动，更新 icache
    flush_icache_range(vectors, vectors + PAGE_SIZE);
   …
}
```

由此可见在 arch/arm/kernel/entry-armv.S 准备了异常向量表及其跳转地址，而实际工作时还需要再做一次移动，0XFFFF0000 附近才是它们的工作地点。

2. SMP 中异常的初始化

对于 SMP 系统，发生异常时每个处理器都要索引到自己的异常表，否则处理器是没法工作的。在 CPU0 的初始化函数 void __init early_trap_init(void)已经建立了虚拟地址与异常表映射，接下来将其余处理器的启动过程中 CPU0 建立的那套页表页目录复制过来这样就可以进行异常索引了。

在 CPU0 启动过程中，会执行 int __cpuinit __cpu_up(unsigned int cpu)。下面代码为其余 CPU 复制页表页目录：

```
int __cpuinit __cpu_up(unsigned int cpu)
{
    …
/*CPU0 复制自己的页表页目录，这样在 4.1.3 节里 CPU0 为自己创建的异常表也为其他 CPU 可用了*/
    pgd = pgd_alloc(&init_mm);
    …
    secondary_data.pgdir = virt_to_phys(pgd);
    …

    /*
```

```
        * Now bring the CPU into our world.
        */
       ret = boot_secondary(cpu, idle);
       …
}
```

由此可以得出结论：在一个 SMP 系统里，在 CPU0 初始化异常表之后，所有其他的 CPU 都共享这个异常入口，地址都一致。

2.2　中断体系

2.2.1　Cortex A9 多核处理器的中断控制器 GIC

在 CA9 以前，每种 SOC 的中断控制器是自己实现的，但是到了 CA9 SMP 以后，中断控制器成为 ARM 规范的一部分，各家的处理器都遵循 ARM 中断控制器 GIC 规范——IHI0048A_gic_architecture_spec。其中原因在于，对于非 SMP 的架构，中断控制器就是控制中断的功能，完成中断的记录、ack、屏蔽等功能即可，各家厂商爱怎么玩就怎么玩。但是到了 SMP 结构下，除了 ack、记录、屏蔽之外，中断控制器还面临如下问题。

（1）中断分发给哪颗处理器。

（2）处理器间通信如何实现——处理器通信本质上就是一种中断。

（3）处理器局部中断。

ARM 提供的中断控制器 GIC 就是解决以上问题，根据 GIC 规范，中断安排如下（参见 IHI0048A_gic_architecture_spec 的 Figure 2-1 GIC logical partitioning into Distributor and CPU interfaces）。

（1）32-1019 SPI　全局中断，可指定分发到不同的处理器。

（2）16-31 PPP　处理器局部中断，处理器局部可见。

（3）0-15 SGI　软件产生中断，由 CPU 写 ICDSGIR 产生，可指定分发到不同的处理器。

GIC 本身分成两部分：

（1）GIC 的 DIST 部分，这部分是所有 CPU 公用的。

（2）每颗 CPU 自己 CPU　interface 部分，这部分控制寄存器地址，只能被 CPU 局部可见，ARM 推荐各 SOC 厂商将该地址实现为相同位置。

以上概念在 GIC 描述结构 struct gic_chip_data 中得到体现。

```
struct gic_chip_data {
    unsigned int irq_offset;    //中断号的偏移量
    void __percpu __iomem **dist_base; //记录每颗 CPU 访问 GIC Dist 的基地址
    void __percpu __iomem **cpu_base; /*记录每颗 CPU 访问 GIC CPU interface
的基地址，每颗处理器看到同样的地方，但是访问的却是不同的寄存器*/
   …
    unsigned int gic_irqs;
```

```
};
```

2.2.2 MT6577 的中断体系

MTK 与 Android 是绝配。在智能机时代， MTK 尽管开始走了段弯路，但是很快拨乱反正。MTK 的智能机时代策略，与其功能机时代如出一辙，这与其市场定位有关。

MTK 手机处理器的技术路线不追求最快，但一定会在市场需要时准时出现；不追求最强，但一定能流畅运行最新版本的 Android 系统与应用。

MTK 的公板，从器件选型、电路板设计到 Android 系统实现，MTK 亲力亲为，几乎做好了所有技术上的工作。MTK 的参考设计总是能做到与大陆手机产业链完美匹配，着力支持手机供应链中最常用、供货最有保障器件和外设，而同一种功能器件和外设，进行多家验证，以确保整机商供应链的安全。由此，华南的 IC、模具、LCD、SMT 等产业链以 MTK 马首是瞻，总是自觉地将产品与 MTK 公板的兼容性作为其重要工作。

本节分析 MT6577 的中断体系。MT6577 的 Datasheet 不是个公开的文档，况且该文档里也并没有清晰地阐述其中断体系的结构。 但是通过 U8836d 公开的代码却可以完整地分析出 MT6577 的中断架构。MT6577 的中断体系的最高层依然遵循 GIC 规范，可参见 ARM 公司 release 相关技术文件。MT6577 中的实现可参见 U8836d 公开的代码中的文件：

```
//U8836D\mediatek\platform\mt6577\kernel\core\include\mach\mt6577_irq.c

//16 个软件中断
#define NR_GIC_SGI              16
//16 个处理器局部中断，在 mt6577 中其实只使用了 5 个，从第 27 号开始
#define NR_GIC_PPI              16
//128 个全局中断
#define MT6577_NR_SPI            (128)
#define NR_MT6577_IRQ_LINE        (NR_GIC_SGI + NR_GIC_PPI + MT6577_NR_SPI)
//全局中断从第 32 号开始
#define GIC_PRIVATE_SIGNALS       32
//处理器局部中断的起始号
#define GIC_PPI_OFFSET            (27)
//如下定义了 5 个处理器局部中断
#define GIC_PPI_GLOBAL_TIMER      (GIC_PPI_OFFSET + 0)
#define GIC_PPI_LEGACY_FIQ        (GIC_PPI_OFFSET + 1)
#define GIC_PPI_PRIVATE_TIMER     (GIC_PPI_OFFSET + 2)
#define GIC_PPI_WATCHDOG_TIMER    (GIC_PPI_OFFSET + 3)
#define GIC_PPI_LEGACY_IRQ        (GIC_PPI_OFFSET + 4)
```

然后从 32 号中断开始定义了 104 个全局中断。

```
#define MT6577_L2CCINTR_IRQ_ID              (GIC_PRIVATE_SIGNALS + 0)
…
#define MT6577_USB0_IRQ_ID                 (GIC_PRIVATE_SIGNALS + 8)
```

```
#define MT6577_USB1_IRQ_ID              (GIC_PRIVATE_SIGNALS + 9)
…
#define MT6577_EINT_IRQ_ID              (GIC_PRIVATE_SIGNALS + 96)
…
#define MT6577_EINT_DIRECT7_IRQ_ID       (GIC_PRIVATE_SIGNALS + 104)
```

以上中断体系的实现是通过 GIC 的配置来完成，而 GIC 的配置在初始化过程中完成。GIC 初始化包括 DIST 的初始化和每颗 CPU 自己 interface 的初始化。DIST 初始化由 CPU0 完成，然后 CPU0 再完成自己备份的 interface 初始化。在 CPU1 起来之后再完成 CPU1 的相关 interface 初始化。

CPU0 的工作代码如下：

```
void __init mt_init_irq(void)
{   …
    //DIST 的初始化
    mt_gic_dist_init();
    //CPU0 的 interface 初始化
    mt_gic_cpu_init();
}

static void mt_gic_dist_init(void)
{
    unsigned int i;
    u32 cpumask = 1 << smp_processor_id();
    cpumask |= cpumask << 8;
    cpumask |= cpumask << 16;

    writel(0, GIC_DIST_BASE + GIC_DIST_CTRL);

    /*
    从 32 号开始，设置 SPI 中断电平触发，N-N 模式。GIC 从 GIC_DIST_CONFIG --0xC00
    开始是中断分发寄存器 ICDICFR，每个 SPI 对应 2 位
    */
    for (i = 32; i < (MT6577_NR_SPI + 32); i += 16) {
       writel(0, GIC_DIST_BASE + GIC_DIST_CONFIG + i * 4 / 16);
    }

    /*
    所有 SPI 分发给 CPU0。这部分代码只有 CPU0 才能执行到。GIC 从 GIC_DIST_TARGET
    --0x800 开始是中断分发寄存器，长度根据支持的 SPI 不同而不同，每个中断对应 8 位，哪
    一位置 1，该中断就发到哪颗处理器。Cpumask 每 8 位的最低位都置 1，所以从 32 号中断开
    始，SPI 都分发 CPU0
    */
    for (i = 32; i < (MT6577_NR_SPI + 32); i += 4) {
       writel(cpumask, GIC_DIST_BASE + GIC_DIST_TARGET + i * 4 / 4);
    }
```

```
    /*
    GIC 从 GIC_DIST_PRI --0x400 开始是中断优先级寄存器，长度根据支持的 SPI 不同而不同，每个中断对应 8 位，这里把所有 SPI 优先级都设置为相同优先级
    */
    for (i = 32; i < NR_MT6577_IRQ_LINE; i += 4) {
       writel(0xA0A0A0A0, GIC_DIST_BASE + GIC_DIST_PRI + i * 4 / 4);
    }

    /*
    GIC 从 GIC_DIST_ENABLE_CLEAR --0x180 开始是中断屏蔽寄存器 ICDICER，长度根据支持的 SPI 不同而不同，每个中断对应 1 位，置 1 屏蔽该中断
    */
    for (i = 32; i < NR_MT6577_IRQ_LINE; i += 32) {
       writel(0xFFFFFFFF, GIC_DIST_BASE + GIC_DIST_ENABLE_CLEAR + i * 4 / 32);
    }
    /*
    把挂到 Linux 中断描述数组里去，每个中断一个 struct irq_desc，以前是个数组，现在可选成 radix 树。每个中断一个 struct irq_desc，记录两个中断处理的关键数据结构 struct irq_chip 和 irq_flow_handler_t。前者是用来对付中断控制器，在 mt6577 上是 struct irq_chip mt_irq_chip，后者是中断处理程序，分成 void handle_level_irq(unsigned int irq, struct irq_desc *desc)，其用来对付电平触发和 void handle_edge_irq(unsigned int irq, struct irq_desc *desc)，其用来对付边沿触发
    */
    for (i = GIC_PPI_OFFSET; i < NR_MT6577_IRQ_LINE; i++) {
       irq_set_chip_and_handler(i, &mt_irq_chip, handle_level_irq);
       set_irq_flags(i, IRQF_VALID | IRQF_PROBE);
    }
#ifdef CONFIG_FIQ_DEBUGGER
    irq_set_chip_and_handler(FIQ_DBG_SGI,                    &mt_irq_chip,
    handle_level_irq);
    set_irq_flags(FIQ_DBG_SGI, IRQF_VALID | IRQF_PROBE);
#endif

    /*
    GIC 从 GIC_ICDISR --0x80 开始是中断屏蔽寄存器 ICDISR，长度根据支持的 SPI 不同而不同，每个中断对应 1 位，置 0 表示该中断是安全中断，置 1 表示该中断是非安全中断，这里全置 1
    */
    for (i = 32; i < NR_IRQS; i += 32)
    {
       writel(0xFFFFFFFF, GIC_ICDISR + 4 * (i / 32));
    }
```

```
    /*
    ICDDCR 位于 0x，中断总开关打开
    */
   writel(3, GIC_DIST_BASE + GIC_DIST_CTRL);
}

//该函数在每颗 CPU 被激活时得到执行
static void mt_gic_cpu_init(void)
{
    int i;

    /*
    关闭 PPI，打开 SGI。PPI 需要用到时调用 void mt_enable_ppi(int irq)打开，在 u8836d
    系统里只有 local timer 是 PPI 中断，在其初始化时将自己对应使能位打开，SGI 必须全
    打开，SGI 用来做处理器间中断的基础
    */
    writel(0xffff0000, GIC_DIST_BASE + GIC_DIST_ENABLE_CLEAR);
    writel(0x0000ffff, GIC_DIST_BASE + GIC_DIST_ENABLE_SET);

    /* 设置 PPI SGI 中断优先级为 0x80*/
    for (i = 0; i < 32; i += 4)
       writel(0x80808080, GIC_DIST_BASE + GIC_DIST_PRI + i * 4 / 4);

    /*设置 PPI SGI 中断优先为非 secure 中断*/
    writel(0xFFFFFFFF, GIC_ICDISR);

    /*操作自己的 CPU INTERFACE 的 ICCPMR 寄存器，只有优先级高于 0xF0 的中断才会送入
    本处理器*/
    writel(0xF0, GIC_CPU_BASE + GIC_CPU_PRIMASK);

/*操作自己的 CPU INTERFACE 的 ICCICR 寄存器，做如下设置：
（1）允许向本处理器送 secure 中断
（2）允许向本处理器送非 secure 中断
（3）把 secure 中断送到本处理器的 FIQ 中断线
（4）中断抢占通过 secure binary point register 判断
（5）一个 ICCIAR 的 secure 读，导致非 secure 最高优先级 pending 中断的 ack
*/

    writel(0x1F, GIC_CPU_BASE + GIC_CPU_CTRL);

    dsb();
}
```

值得注意的是 MT6577 全局中断的实现又分为两种。

（1）对于集成在 SOC 里的外设，直接对应一个全局中断，如 MT6577_USB1_IRQ_ID，

在其驱动初始时直接将其中断处理函数注册到内核的中断处理里数组中：

```
request_irq(nIrq, musbfsh->isr, IRQF_TRIGGER_LOW, dev_name(dev), musbfsh)
```

（2）对于位于 SCO 之外的外设，则通过 MT6577_EINT_IRQ_ID 进行转发，MT6577 在内核里准备了一个外设中断处理函数的数组：

```
typedef struct
{
    void (*eint_func[EINT_MAX_CHANNEL])(void);
    unsigned int eint_auto_umask[EINT_MAX_CHANNEL];
} eint_func;
```

这些外设通过 void mt65xx_eint_registration(…)注册自己的中断函数。

比如触屏控制器 gt813，mediatek\custom\out\huaqin77_cu_ics2\kernel\touchpanel\Gt813_driver.c 通过该函数注册自己的中断处理函数：

```
mt65xx_eint_registration(…, tpd_eint_interrupt_handler, 1);
```

当这些 SOC 外设中断发生以后，MT6577 将在 GIC 的 MT6577_EINT_IRQ_ID 上产生中断，从而触发其中断处理函数 static irqreturn_t mt65xx_eint_isr(int irq, void *dev_id)。在该函数里将检查 eint_func 数组，进一步触发对应的处理函数。

2.2.3 Exynos4 的中断体系

三星，ARM 处理器界的新王者，近年来抢先实现每一代 ARM 处理器，而且通过手机处理器与其庞大硬件产业链的有机整合成为 Apple 的最有力对手。

第一次将三星与高性能手机处理器联系起来的是其 S5PV210，凭借超出同级处理器一倍的 L2 Cache，S5PV210 成为当时跑得最快的 CA8 ARM。紧接着三星开始抢跑 ARM 界，在 CA9 时期，三星已经成为 ARM 处理器界的第一阵营，在 CA15 时期三星成为第一家先进 ARM 架构的实现者。

三星的 ARM 处理器成功背后的很大原因在于三星强大的产业链整合能力。在前端，三星手机的攻城略地保证了三星处理器出货量。在后端，三星的半导体工厂有比肩 Intel 的能力，总是可以用更先进的工艺来降低功耗，节省成本。在市场大门开启的时候，三星的对手却要为处理器代工厂的工艺和良率伤透脑筋，为炫酷的 LCD 屏幕供货周期头疼，为了内存和 EMMC 的价格波动心惊胆战，而所有这些，三星集团的各个工厂数月之前都已经协调完毕，正按计划批量出货了。

1. exynos4 的中断体系

CPU0 的 GIC 初始化工作从 void __init exynos4_init_irq(void)开始。

```
void __init exynos4_init_irq(void)
{
     int irq;

     gic_bank_offset = soc_is_exynos4412() ? 0x4000 : 0x8000;
```

```
/*S5P_VA_GIC_DIST 是 CPU0 的 GIC DIST 基地址，S5P_VA_GIC_CPU 是 CPU0 的 GIC CPU
interface 基地址*/
    gic_init(0, IRQ_PPI_MCT_L, S5P_VA_GIC_DIST, S5P_VA_GIC_CPU);
    …

}

void __init gic_init(unsigned int gic_nr, unsigned int irq_start,
    void __iomem *dist_base, void __iomem *cpu_base)
{
    struct gic_chip_data *gic;
    int cpu;

    …
    //GIC 描述结构：struct gic_chip_data
    gic = &gic_data[gic_nr];
//为每颗 CPU 分配放置 DIST 和 CPU  interface 基地址的内存地址
    gic->dist_base = alloc_percpu(void __iomem *);
    gic->cpu_base = alloc_percpu(void __iomem *);
    …
    //先初始化 CPU0 的 DIST 和 CPU  interface 基地址
    for_each_possible_cpu(cpu) {
        *per_cpu_ptr(gic->dist_base, cpu) = dist_base;
        *per_cpu_ptr(gic->cpu_base, cpu) = cpu_base;
    }

    gic->irq_offset = (irq_start - 1) & ~31;

    if (gic_nr == 0)
        gic_cpu_base_addr = cpu_base;

    //DIST 的初始化
    gic_dist_init(gic, irq_start);
//CPU0 的 CPU  interface 基地址的内存
    gic_cpu_init(gic);
}

/*所谓 GIC dist 的初始化，系统里只在 CPU0 进行，核心工作是配置 GIC SPI 中断（通常外围
设备产生的中断）*/
static void __init gic_dist_init(struct gic_chip_data *gic,
    unsigned int irq_start)
{
    unsigned int gic_irqs, irq_limit, i;
    void __iomem *base = gic_data_dist_base(gic);
    u32 cpumask = 1 << smp_processor_id();
```

```
//cpumask: 每个 8 位的最低位都置 1

    cpumask |= cpumask << 8;
    cpumask |= cpumask << 16;

    writel_relaxed(0, base + GIC_DIST_CTRL);

    /*
     * Find out how many interrupts are supported.
     * The GIC only supports up to 1020 interrupt sources.
     */
    gic_irqs = readl_relaxed(base + GIC_DIST_CTR) & 0x1f;
    gic_irqs = (gic_irqs + 1) * 32;
    if (gic_irqs > 1020)
        gic_irqs = 1020;

    /*
     * Set all global interrupts to be level triggered, active low.
     */
//先将 SPI 设置为电平触发

    for (i = 32; i < gic_irqs; i += 16)
        writel_relaxed(0, base + GIC_DIST_CONFIG + i * 4 / 16);

    /*
     * Set all global interrupts to this CPU only.
     */
/*偏移量 0x800~0x81C 空间的每个 BYTE 指出了对应的中断分发到哪颗 CPU 上，这里把所有的全局中断的分发方向都指向自己，这时只有 CPU0 可用。以后还可以使用 struct irq_chip 的 int (*set_affinity)(unsigned int irq,const struct cpumask *dest);函数将中断绑定到另外的 CPU 上*/

    for (i = 32; i < gic_irqs; i += 4)
        writel_relaxed(cpumask, base + GIC_DIST_TARGET + i * 4 / 4);

    /*
     * Set priority on all global interrupts.
     */
//0x400-0x7F8

    for (i = 32; i < gic_irqs; i += 4)
        writel_relaxed(0xa0a0a0a0, base + GIC_DIST_PRI + i * 4 / 4);

    /*
```

```
     关闭除 PPI 和 SGIs 之外的中断
     */
    for (i = 32; i < gic_irqs; i += 32)
        writel_relaxed(0xffffffff, base + GIC_DIST_ENABLE_CLEAR + i * 4 / 32);

    /*
     * Limit number of interrupts registered to the platform maximum
     */
    irq_limit = gic->irq_offset + gic_irqs;
    if (WARN_ON(irq_limit > NR_IRQS))
        irq_limit = NR_IRQS;

    /*
     * Setup the Linux IRQ subsystem.
     */
//填充 linux 里 generic 中断分发数组
    for (i = irq_start; i < irq_limit; i++) {
        irq_set_chip_and_handler(i, &gic_chip, handle_fasteoi_irq);
        irq_set_chip_data(i, gic);
        set_irq_flags(i, IRQF_VALID | IRQF_PROBE);
    }

    writel_relaxed(1, base + GIC_DIST_CTRL);
}

/*CPU0 的 GIC cpu interface 中断初始化，主要是 PPI 和 SGI 初始化，SGI 是处理器间通信
的重要手段*/
static void __cpuinit gic_cpu_init(struct gic_chip_data *gic)
{
    void __iomem *dist_base = gic_data_dist_base(gic);
    void __iomem *base = gic_data_cpu_base(gic);
    int i;

    /*
     * Deal with the banked PPI and SGI interrupts - disable all
     * PPI interrupts, ensure all SGI interrupts are enabled.
     */
//值得注意的是，这里 PPI 中断是被屏蔽的
    writel_relaxed(0xffff0000, dist_base + GIC_DIST_ENABLE_CLEAR);
    writel_relaxed(0x0000ffff, dist_base + GIC_DIST_ENABLE_SET);

    /*
     * Set priority on PPI and SGI interrupts
     */
    for (i = 0; i < 32; i += 4)
```

```
        writel_relaxed(0xa0a0a0a0, dist_base + GIC_DIST_PRI + i * 4 / 4);

    writel_relaxed(0xf0, base + GIC_CPU_PRIMASK);
    writel_relaxed(1, base + GIC_CPU_CTRL);
}
```

2. CPUX 相关的 GIC 配置

CPUX 不需要初始化 GIC 的 SPI 中断，所以 CPUX 相关的 GIC 初始化工作完成自身的 GIC cpu interface 中断初始化即可。值得关注是其执行流程：

```
asmlinkage void __cpuinit secondary_start_kernel(void)
{…
platform_secondary_init(cpu);
…}


void __cpuinit platform_secondary_init(unsigned int cpu)
{
/*显然对于 Exynos4x12 处理器，CPUX 的 GIC DIST 和 CPU interface 基地址的计算是：
CPU0 的对应基地址+ 前面 CPU 占用的长度*/
    void __iomem *dist_base = S5P_VA_GIC_DIST +
                 (gic_bank_offset * cpu);
    void __iomem *cpu_base = S5P_VA_GIC_CPU +
                (gic_bank_offset * cpu);

    gic_secondary_init_base(0, dist_base, cpu_base);

…
}

void __cpuinit gic_secondary_init_base(unsigned int gic_nr,
                      void __iomem *dist_base,
                      void __iomem *cpu_base)
{
//先把基地址放到 GIC 描述结构里的数组里
    if (dist_base)
        *__this_cpu_ptr(gic_data[gic_nr].dist_base) = dist_base;
    if (cpu_base)
        *__this_cpu_ptr(gic_data[gic_nr].cpu_base) = cpu_base;
//CPUX 的 PPI SGI 中断初始化见前面的介绍
    gic_cpu_init(&gic_data[gic_nr]);
}
```

2.2.4 OMAP4 的中断体系

尽管已经宣布退出手机市场，但是作为移动处理器领域的领袖， Ti 在相当长的时间

里总是抢先发布性能最强的新一代 ARM 处理器，而且早期还会搭配其强劲的 DSP，以配合 ARM CORE 工作。尽管 Ti 在 3G 时代遭受专利困境，但是凭借其强大的 ARM 处理器设计能力在没有 Modem 的情况下支撑了两代：Omap3 是第一款 Cortex A8 产品，且加入了 C64+DSP，当时的性能在业界无出其右者。接着随着 Neon 和硬件编解码的兴起，Omap4 弱化了 DSP 作用，但依然领导业界进入双核时代。

也许因为 Omap5 的功耗实在不能满足手机需求，也许无法集成 Modem 弱化了 Ti 的竞争力，也许 Ti 早已决心转身模拟，以后的手机处理器将不会有 Ti 的身影。Ti 将 Omap5 的积累转化成其嵌入式产品 keystoneII，但是嵌入式处理器的演进由于其市场需求的原因，将不会像手机处理器那样精彩。

本节介绍 Omap4 中断体系。

1. 初始化

```
void __init gic_init_irq(void)
{
    …

    //首先 map 出中断控制器 Distributor 寄存器空间
    gic_dist_base_addr = ioremap(OMAP44XX_GIC_DIST_BASE, SZ_4K);
    BUG_ON(!gic_dist_base_addr);
    //对 Distributor 的初始化
    gic_dist_init(0, gic_dist_base_addr, 29);

    /* Static mapping, never released */
    //map 出中断控制器 cpu interface 寄存器空间

    gic_cpu_base_addr = ioremap(OMAP44XX_GIC_CPU_BASE, SZ_512);
    BUG_ON(!gic_cpu_base_addr);
    gic_cpu_init(0, gic_cpu_base_addr);

/*以上 2 次 map，实际上就是在页表中建立虚拟地址到物理地址的映射。这个动作只在主处理器初始化时进行，而在第二颗处理器初始化时并未进行 map 操作。其原因在于：
（1）第二颗处理器初始化过程中上拷贝了主处理器的页表页目录。
（2）从每颗处理器往外看，Distributor 系统中就一个，其物理地址位于：
#define OMAP44XX_GIC_DIST_BASE      0x48241000
虽然 cpu interface 每个处理器都有自己的空间，但是显然在 Omap4 的实现中将其做成了相同的物理地址*/
#define OMAP44XX_GIC_CPU_BASE       0x48240100

}

void __cpuinit gic_cpu_init(unsigned int gic_nr, void __iomem *base)
{   …
```

```
    gic_data[gic_nr].cpu_base = base;

/*初始化 CPU 中断控制信息，这里没有显示的区分对哪颗 CPU 操作，但是由于上述的虚拟地址的拷贝关系，哪个处理器执行到这里就对哪颗处理器操作*/
    writel(0xf0, base + GIC_CPU_PRIMASK);
    writel(1, base + GIC_CPU_CTRL);
}

void __init gic_dist_init(unsigned int gic_nr, void __iomem *base, unsigned int irq_start)
{

//这里最关键的是记下 Distributor 的虚拟地址
    gic_data[gic_nr].dist_base = base;
}
```

2. 第二颗 CPU interface 的初始化

```
void __cpuinit platform_secondary_init(unsigned int cpu)
{
    trace_hardirqs_off();


//初始化该 CPU 中断控制信息
    gic_cpu_init(0, gic_cpu_base_addr);

    ...
    spin_lock(&boot_lock);
    spin_unlock(&boot_lock);
}
```

3. CPU 间中断发射

```
static inline void smp_cross_call(const struct cpumask *mask)
{
    gic_raise_softirq(mask, 1);
}

//mask 记录了要向哪些处理器发射中断，irq 记录要产生的中断号

#ifdef CONFIG_SMP
void gic_raise_softirq(const struct cpumask *mask, unsigned int irq)
{
    unsigned long map = *cpus_addr(*mask);

    /* this always happens on GIC0 */
```

```
/*Distributor 的 Software Generated Interrupt Register (ICDSGIR)用法如下：
第 16 位——第 23 位 ： 目标处理器
第 0 位——第 4 位  ：  中断号
以下语句向 mask 标记的 CPU 发射 irq 号中断*/
writel(map << 16 | irq, gic_data[0].dist_base + GIC_DIST_SOFTINT);
}
#endif
```

2.3 中断处理

中断表面的作用是作为硬件和驱动的一部分，把中断送入处理器，激活中断 handler 以响应外设。但是对于内核中断的意义远非这些，对于调度器，中断是主要激励源，中断的发生意味着可能的线程唤醒，中断退出意味着可能的线程抢占。对于信号机制，若发生在用户态，中断返回意味着信号的执行。对于多核处理器中断是其处理器间交流的手段。

本书把中断在内核其他机制的分析放在其相关章节，这里仅仅分析中断本身。

2.3.1 中断的基本结构

为了有一个整体的印象，先来分析中断基本结构，以内核模式发生中断为背景。首先中断异常代码识别是 USER 模式还是内核模式，如果是内核模式发生的中断，则跳入 __irq_svc：

```
__irq_svc:
    svc_entry //中断进入
#ifdef CONFIG_PREEMPT //抢占维护工作
    //得到 thread_info 指针
    get_thread_info tsk
    /* preempt_count 的最低 8 位记录着是否可以线程抢占，中断发生时可以发生中断嵌套，
    更高优先级中断抢占当前级别的中断。但是新唤醒的线程却不能在当前中断处理过程中抢占中
    断 handler。因为当前中断借用当前线程的内核栈，优先级是针对线程而言的，即使新唤醒
    的线程优先级较高也是比这个当前线程高，并不能说明比中断本身高。再者线程抢占中断是没
    有意义的。若对实时性要求的确很高，可以把中断处理线程化，中断唤醒线程化的 handler
    后即退出，让线程去抢占线程*/
    ldr r8, [tsk, #TI_PREEMPT]      @ get preempt count
    add r7, r8, #1          @ increment it
    str r7, [tsk, #TI_PREEMPT]
#endif

    irq_handler  //中断分发
#ifdef CONFIG_PREEMPT//抢占维护工作
    //本次中断退出，恢复抢占计数
    str r8, [tsk, #TI_PREEMPT]      @ restore preempt count
```

```
    ldr r0, [tsk, #TI_FLAGS]      @ get flags
    teq r8, #0                    @ if preempt count != 0
    movne   r0, #0                @ force flags to 0
    //更高优先级的线程被唤醒，若当前线程被抢占
    tst r0, #_TIF_NEED_RESCHED
    blne svc_preempt
#endif
    ldr r4, [sp, #S_PSR]          @ irqs are already disabled
    …
    //中断退出
    svc_exit r4                   @ return from exception
 UNWIND(.fnend      )
ENDPROC(__irq_svc)
```

svc_entry 是一个宏，用于 SVC 模式中断进入，主要是完成寄存器保护的工作：

```
    .macro  svc_entry, stack_hole=0
    ...
    //把内核栈的指针拉下来，为存放寄存器留出空间
    sub sp, sp, #(S_FRAME_SIZE + \stack_hole - 4)
    …
    //把 r1-r12 这 12 个寄存器放进内核栈
    stmia   sp, {r1 - r12}
    …
    .endm
```

以上是内核态发生中断的处理，对于用户态发生中断，最大区别是在返回会检查是否有 signal 要处理。接下来继续看中断如何进入中断 handler。

中断 handler 的入口也是一个宏定义：

```
.macro  irq_handler
…
    arch_irq_handler_default
9997:
    .endm
```

arch_irq_handler_default 的实现位于 <asm/entry-macro-multi.S>，在文件 kernel/arch/arm/kernel/entry-armv.S 里包含了该文件：

```
    .macro  arch_irq_handler_default
    //把 GIC 基地址取出来放到 r5 里
    get_irqnr_preamble r5, lr
1:  get_irqnr_and_base r0, r6, r5, lr
    movne   r1, sp
    @
    @ routine called with r0 = irq number, r1 = struct pt_regs *
    @
```

```
    adrne   lr, BSYM(1b)
    /*普通中断，分发*/
    bne asm_do_IRQ

#ifdef CONFIG_SMP
    //检查是否是 IPI 中断，IPI 中断是 SGI 类型，中断号小于 16
    ALT_SMP(test_for_ipi r0, r6, r5, lr)
    ALT_UP_B(9997f)
    //r1 存放栈指针，跟 r0 一道作为参数送入 do_IPI
    movne   r1, sp
    adrne   lr, BSYM(1b)
    /*处理器间中断 */
    bne do_IPI

#ifdef CONFIG_LOCAL_TIMERS
    //检查处理器局部时钟，29 号
    test_for_ltirq r0, r6, r5, lr
    movne   r0, sp
    adrne   lr, BSYM(1b)
    /*tick 中断*/
    bne do_local_timer
#endif
…

#endif
9997:
    .endm
```

2.3.2　中断源识别

中断发生时，首先要确认该中断来自系统中哪个部位，这就是中断源识别。但是尽管中断控制器在 CA9 成为规范，但是每种架构的中断体系都不一样，所以中断源识别在每种架构下的实现也都不一样，本节以 exynos4 与 MT6577 为例分析中断源识别。

1. exynos4 的中断源实现

```
//代码位置：arch/arm/mach-exynos/include/mach/entry-macro.S

        .macro  get_irqnr_preamble, base, tmp
#ifdef CONFIG_ARCH_EXYNOS4
        mov \tmp, #0
//读取 CP15 里的 MPIDR 寄存器
        mrc p15, 0, \base, c0, c0, 5
// MPIDR 的最低 2 位指示当前读取动作是哪颗 CPU 所为，寄存器 base 里就是处理器号
        and \base, \base, #3
```

```
        cmp \base, #0
        beq 1f
        ldr \tmp, =gic_bank_offset
        ldr \tmp, [\tmp]
        cmp \base, #1
        beq 1f
        cmp \base, #2
//CPU2
        addeq   \tmp, \tmp, \tmp
//CPU3
        addne   \tmp, \tmp, \tmp, LSL #1
#endif
1:      ldr \base, =gic_cpu_base_addr
        ldr \base, [\base]

#ifdef CONFIG_ARCH_EXYNOS4
        add \base, \base, \tmp
#endif
        .endm
```

值得注意的是，该版本的代码源自 SAMSUNG 公开的其 i9300（四核处理器版本）源码包，可见其 GIC 的 cpu interface 实现没有遵循 ARM 的官方推荐——每颗处理器 cpu interface 基地址都不一样。而对于 kernel.org，仅支持 exy4210（双核版本），代码中其 GIC 的 cpu interface 实现又遵循了 ARM 官方的推荐——每颗处理器 cpu interface 基地址都一样。

```
    .macro  get_irqnr_preamble, base, tmp
        //直接取基地址，两颗处理器得到的地址都一样
        ldr \base, =gic_cpu_base_addr
        ldr \base, [\base]
        .endm

.macro  get_irqnr_and_base, irqnr, irqstat, base, tmp

/*base 是当前 CPU GIC interface 的基地址，基地址偏移#GIC_CPU_INTACK （0XC）位置
的寄存器是 Interrupt Acknowledge Register (ICCIAR)。它的 12～10 位记录了如果是
SGI 类型的中断，产生该中断的 CPU 号。第 0～9 位记录的就是当前中断号*/

        ldr     \irqstat, [\base, #GIC_CPU_INTACK] /* bits 12-10 = src CPU,
9-0 = int # */

        cmp     \irqnr, #29
        cmpcc   \irqnr, \irqnr
        cmpne   \irqnr, \tmp
        cmpcs   \irqnr, \irqnr
        addne   \irqnr, \irqnr, #32
…
```

```
        .endm
```

2. MT6577 的中断源实现

```
//取出当前处理器 gic intrface 的基地址
.macro  get_irqnr_preamble, base, tmp
    ldr \base, =GIC_CPU_BASE
.endm

.macro get_irqnr_and_base, irqnr, irqstat, base, tmp
  //取 ICCIAR 到 irqstat
    ldr \irqstat, [\base, #GIC_CPU_INTACK] /* bits 12-10 = src CPU, 9-0 =
int # */
// NR_IRQS 是 mt6577 支持的最高中断号 16+16+128
    ldr \tmp, =NR_IRQS
    //将 ICCIAR 与上 0x1c00 的反码，取 9-0 位到 irqnr
    bic \irqnr, \irqstat, #0x1c00

    /* if (irqnr >= NR_IRQS) return NO_IRQ (0) */
    cmp \irqnr, \tmp
    movcs \tmp, #0
    bcs BSYM(702f)

    /* if (irqnr >= 32) return HAVE_IRQ (1) */
    cmp \irqnr, #(32)
    movcs \tmp, #1
    bcs BSYM(702f)

    /* if (irqnr == FIQ_DBG_SGI) return HAVE_IRQ (1) */
    cmp \irqnr, #FIQ_DBG_SGI
    moveq \tmp, #1
    beq BSYM(702f)

    /* otherwise, return NO_IRQ (0) */
    mov \tmp, #0

702:
    cmp \tmp, #0
    cmpeq \irqnr, \irqnr
.endm

//ipi 类型的中断检测
.macro test_for_ipi, irqnr, irqstat, base, tmp
    //取出中断号放在 irqnr
    bic \irqnr, \irqstat, #0x1c00
    //用中断号减 16
```

```
    cmp \irqnr, #16
    //c 清零，无符号数小于，小于 16 号中断，为 SGI，写 GIC，ack 该中断
    strcc \irqstat, [\base, #GIC_CPU_EOI]
    /*c 置位表示无符号数大于或等于，则不属于 SGI 中断，若小于 16 号中断，该指令不执行，
bne 条件成立，若大于 16 号中断，该指令执行，bne 执行条件不满足*/
    cmpcs \irqnr, \irqnr
.endm

//检测是否是 lcoal timer
.macro test_for_ltirq, irqnr, irqstat, base, tmp
    //取出中断号放在 irqnr
    bic     \irqnr, \irqstat, #0x1c00
    mov     \tmp, #0
    //local timer 是 29 号中断，比较
    cmp     \irqnr, #29
    moveq   \tmp, #1
    /*写 GIC 的 end of interrupt 寄存器，该寄存器格式要求 cpuid 个中断号，ack 该中断，
与 icciar 一样，把 irqstat 回填即可*/
    streq   \irqstat, [\base, #GIC_CPU_EOI]
    //如果是 29，tmp 里不应该是 0
    cmp     \tmp, #0
.endm

//检查 watchdog 中断
.macro test_for_wdtirq, irqnr, irqstat, base, tmp
    //取出中断号放在 irqnr
    bic     \irqnr, \irqstat, #0x1c00
    mov     \tmp, #0
    cmp     \irqnr, #30
    moveq   \tmp, #1
    //写 GIC 的 EOI，ack 该中断
    streq   \irqstat, [\base, #GIC_CPU_EOI]
    cmp     \tmp, #0
.endm
```

2.4 数 据 异 常

数据访问异常和取指异常，都是处理器在转换数据页和代码页时遇到的，是虚拟内存机制的上层触发入口，两者工作机理类似，本节以数据异常为例展开，为下一章节虚拟内存的分析做铺垫。

首先再次审视异常向量表中数据异常相关部分，这是数据异常的内核入口：

```
    vector_stub dabt, ABT_MODE, 8
    /*用户态发生的异常，比如程序里访问某个变量*/
```

```
    .long   __dabt_usr          @  0  (USR_26 / USR_32)
    .long   __dabt_invalid          @  1  (FIQ_26 / FIQ_32)
    .long   __dabt_invalid          @  2  (IRQ_26 / IRQ_32)
  /*内核态发生了异常。内核态正常情况下不可能发生这个异常，除非是遇到了 ex_table，再
者就是 BUG 了*/
    .long   __dabt_svc          @  3  (SVC_26 / SVC_32)
    …
```

另一方面，内核准备了一张处理表用来处理每一种数据异常：

```
static struct fsr_info ifsr_info[] = {
    { do_bad,        SIGBUS,  0,       "unknown 0"              },
    …
    { do_bad,        SIGSEGV, SEGV_ACCERR,   "section access flag fault"
      },
    { do_bad,        SIGBUS,  0,       "unknown 4"              },
    { do_translation_fault, SIGSEGV, SEGV_MAPERR,   "section translation
    fault"     },
    { do_bad,        SIGSEGV, SEGV_ACCERR,   "page access flag fault"},
    { do_page_fault,     SIGSEGV, SEGV_MAPERR,   "page translation fault"},
    { do_bad,        SIGBUS,  0,       "external abort on non-linefetch"  },
    { do_bad,        SIGSEGV, SEGV_ACCERR,   "section domain fault"},
    { do_bad,        SIGBUS,  0,       "unknown 10"              },
    { do_bad,        SIGSEGV, SEGV_ACCERR,   "page domain fault"        },
    { do_bad,        SIGBUS,  0,       "external abort on translation"   },
    { do_sect_fault, SIGSEGV, SEGV_ACCERR,   "section permission fault"},
    { do_bad,        SIGBUS,  0,       "external abort on translation"   },
    { do_page_fault,     SIGSEGV, SEGV_ACCERR,   "page permission fault"},
    …
    { do_bad,        SIGBUS,  0,       "unknown 31"              },
};
```

该异常表覆盖了 ARM 处理器可能发生的所有的数据异常，但是因为 ARM 的有些机制 Linux 内核并没使用到，所以对于这类异常只能默认使用 do_bad，这将导致系统 panic。而对于 Linux 使用的 ARM 机制这些异常都有对应的处理函数。数据异常发生后，无论用户态异常还是内核态异常，都会进入到内核的数据异常处理函数中：

```
/*该函数的参数 addr 记录了异常产生的虚拟地址，参数 fsr 记录了异常产生的原因*/
asmlinkage void __exception
do_DataAbort(unsigned long addr, unsigned int fsr, struct pt_regs *regs)
{
    struct thread_info *thread = current_thread_info();
    /*根据异常产生的原因索引处理函数表*/
    const struct fsr_info *inf = fsr_info + fsr_fs(fsr);
    int ret;
    struct siginfo info;
```

```
    //调用合适的异常处理函数，如果返回非零，说明发生了不正常的异常
    if (!inf->fn(addr, fsr & ~FSR_LNX_PF, regs))
        return;

    info.si_signo = inf->sig;
    info.si_errno = 0;
    info.si_code  = inf->code;
    info.si_addr  = (void __user *)addr;
    /*如果是用户态发生的不正常异常，发信号杀死该进程；如果是内核态发生的不正常异常，
panic*/
    arm_notify_die("", regs, &info, fsr, 0);
}
```

接下来就到了内核的虚拟内存处理，这是一个庞大的机制，影响到系统的方方面面，将在本书的其他章节进行分析。

2.5 处理器间通信

处理器间通信的基础是中断，在 CA9 架构下 0～15 号的软件触发中断 SGI。在 GCI 初始化的时候各处理器开放了自己的 0～15 中断，以便接受别的处理器发给自己的中断。而 GIC 的 ICDSGIR 用来产生 SGI 中断，其格式如下：

- Bit 0～3 SGIINTID，产生的 SGI 中断号。
- Bit 4～14 reserved。
- Bit 15 安全位　0：发射安全中断 1：发射非安全中断。
- Bit 16～23 目标处理器位图　每颗处理器对应一位，置 1 将向该处理器发射 SGI 中断。
- Bit 24～25　00：不考虑每个 CPU interface 的指示来发射中断；01 或 10：根据每个 CPU interface 的指示来发射中断或不发射中断。在 ARM Linux 系统系统系统下 bit 24～25 为 00。

```
//接下分析中断代码分析整理自基于开源手机 u8836d (mt6577) 笔记
void irq_raise_softirq(const struct cpumask *mask, unsigned int irq)
{
    unsigned long map = *cpus_addr(*mask);
    int satt;
    u32 val;
/*
向 CPU1 发射 IPI_CPU_START 中断，是 CPU0 wakeup CPU1 的最后一个动作，这里如果检测到
是 IPI_CPU_START 中断，则其安全位被置为 secure 中断。这个中断假设的场景是 CPU1 处在
WFI 状态，发射该中断使其继续往下跑，但是事实上这个中断的并没起到必要的作用。当 CPU0 发
射这个中断时 CPU1 的中断被禁止的，CPU1 直到跑到 void __cpuinit
secondary_start_kernel(void) 的末尾才打开中断，这里并不是 WFI，这里的
```

```
IPI_CPU_START 机制是没有意义的
*/

    satt = (irq == IPI_CPU_START)? 0: (1 << 15);
/*从 dist 偏移量 0x80，记录了每个中断的 secure 属性，每个中断对应 1 位，置 0 表示 secure
中断，置 1 表示非 secure 中断，这里读取要发射中断的属性信息*/
    val = __raw_readl(GIC_ICDISR + 4 * (irq / 32));
    if (!(val & (1 << (irq % 32)))) {   /* secure interrupt? */
    //该位为 0，secure 中断
       satt = 0;
    }
/*将中断号、中断位、中断目标一并写入 ICDSGIR*/
    __raw_writel((map << 16) | satt | irq, GIC_DIST_BASE + 0xf00);

}
```

内核定义了如下几种处理器间中断：

```
enum ipi_msg_type {
    IPI_CPU_START = 1,
    IPI_TIMER = 2,
/*这是最常用的处理器间中断，被调度器使用，它用来通知某目标处理器，其运行队列被挂入若干
task，目标处理器接到该中断后，把外挂队列的 task active 到自己的运行队列，具体参见调度
器一节*/
    IPI_RESCHEDULE,
    IPI_CALL_FUNC,
/*在特定目标处理器上运行某特定函数，这个也很常用，是一种重要的内核机制，在后面有详细介
绍*/
    IPI_CALL_FUNC_SINGLE,
    IPI_CPU_STOP,
    IPI_CPU_BACKTRACE,
};
```

以 IPI_RESCHEDULE 为例分析 IPI 发送过程：

（1）函数指针：static void (*smp_cross_call)(const struct cpumask *, unsigned int);指向处理器间通信的发射函数，在系统初始化时，将该指针指向该平台的 IPI 实现函数，对于 u8836d 自然就是 void irq_raise_softirq(const struct cpumask *mask, unsigned int irq)，而别的 CA9 SMP 架构下的实现与函数也非常相似。

（2）当前处理器侧，调用 void smp_send_reschedule(int cpu)函数，其中 cpu 是目标处理器。

（3）目标处理器侧产生 SGI 中断，导致函数 asmlinkage void __exception_irq_entry do_IPI(int ipinr, struct pt_regs *regs)被调用。从硬件中断的产生到该函数的调用参见上文。

```
//其中 ipinr 是中断号，regs 与普通中断一样是存储的寄存器状态
asmlinkage void __exception_irq_entry do_IPI(int ipinr, struct pt_regs
*regs)
```

```
{
    /*取出 CPU 号*/
    unsigned int cpu = smp_processor_id();
    struct pt_regs *old_regs = set_irq_regs(regs);
    /*中断状态的统计，标志对于 IPI 中断又发生了一次*/
    if (ipinr >= IPI_CPU_START && ipinr < IPI_CPU_START + NR_IPI)
        __inc_irq_stat(cpu, ipi_irqs[ipinr - IPI_CPU_START]);

    switch (ipinr) {
    case IPI_CPU_START:
        /* Wake up from WFI/WFE using SGI */
        break;

    case IPI_TIMER:
        __raw_get_cpu_var(mt_timer_irq) = IPI_TIMER;
        ipi_timer();
        break;
    /*目标处理器接受到信息*/
    case IPI_RESCHEDULE:
    /*调整自己的运行队列*/
        scheduler_ipi();
        break;

    case IPI_CALL_FUNC:
        generic_smp_call_function_interrupt();
        break;
   /*目标侧 IPI_CALL_FUNC_SINGLE 响应*/
    case IPI_CALL_FUNC_SINGLE:
        generic_smp_call_function_single_interrupt();
        break;

        ...
    }
   ...
}
```

IPI_CALL_FUNC_SINGLE

每颗处理器都有一个 struct call_single_queue 队列，队列由“per_cpu(call_single_queue, cpu)”来索引，该队列上每一个节点——struct call_single_data 都有一个函数指针指向某个待该处理器执行的函数。

当前处理器有需要某特定处理器执行函数的时候调用：

```
/*cpu 为目标处理器，func 指向目标处理器需要执行的函数，info 为参数信息 wait 指出是否
等待对方处理器同步*/

int smp_call_function_single(int cpu, smp_call_func_t func, void *info,
```

```
                int wait)
{
    struct call_single_data d = {
        .flags = 0,
    };
    …
    //取出当前 cpu
this_cpu = get_cpu();
    //如果目标处理器就是自己，就不要折腾了，执行即可
    if (cpu == this_cpu) {
        local_irq_save(flags);
        func(info);
        local_irq_restore(flags);
    } else {
/*验证 cpu 号，验证该 cpu 是否在线 */
        if ((unsigned)cpu < nr_cpu_ids && cpu_online(cpu)) {
            struct call_single_data *data = &d;

            if (!wait)
                data = &__get_cpu_var(csd_data);

            //将该 struct call_single_data 的 flags 置 CSD_FLAG_LOCK
csd_lock(data);
           //设置目标处理器位，函数指针信息
            cpumask_set_cpu(cpu, data->cpumask);
            data->func = func;
            data->info = info;
           //执行
            generic_exec_single(cpu, data, wait);
        } else {
            err = -ENXIO;   /* CPU not online */
        }
    }
/*释放该处理器*/
    put_cpu();
    return err;
}

static
int generic_exec_single(int cpu, struct call_single_data *data, int wait)
{
//这里的 cpu 是目标 cpu，索引目标处理器的 struct call_single_queue 队列
struct call_single_queue *dst = &per_cpu(call_single_queue, cpu);
…
```

```
    //关中断，上锁，要保证这个链表操作不被打扰
    raw_spin_lock_irqsave(&dst->lock, flags);
    ipi = list_empty(&dst->list);
    list_add_tail(&data->list, &dst->list);
    //恢复中断，开锁
    raw_spin_unlock_irqrestore(&dst->lock, flags);

    //发送 IPI_CALL_FUNC_SINGLE
    if (ipi)
        arch_send_call_function_single_ipi(cpu);
   //等待，要等待对方处理器把指定函数跑完
    if (wait)
        err = csd_lock_wait(data);
    return err;
}

/*这个等待是瞬间完成的过程，没有休眠*/
static int csd_lock_wait(struct call_single_data *data)
{
    int cpu, nr_online_cpus = 0;
/*检查 CSD_FLAG_LOCK 标志是否被对方清楚，对方处理只有在跑完指定函数后才会清楚这个标
志*/
    while (data->flags & CSD_FLAG_LOCK) {
      //不停检查在线处理器个数
        for_each_cpu(cpu, data->cpumask) {
            if (cpu_online(cpu)) {
                nr_online_cpus++;
            }
        }
      //如果等待的处理器一个都不在了就返回
        if (!nr_online_cpus)
            return -ENXIO;
      //等待操作
        cpu_relax();
    }

    return 0;
}
```

目标 CPU 接到 IPI_CALL_FUNC_SINGLE 后调用：

```
void generic_smp_call_function_single_interrupt(void)
{   //取出当前 struct call_single_queue 队列
    struct call_single_queue *q = &__get_cpu_var(call_single_queue);
    unsigned int data_flags;
```

```
    …
    //如果 struct call_single_queue 队列非空，说明有活要干
    while (!list_empty(&list)) {
        struct call_single_data *data;
        //依次取出待执行的函数指针
        data = list_entry(list.next, struct call_single_data, list);
        list_del(&data->list);

        data_flags = data->flags;
        //执行该函数
        data->func(data->info);

        /*
        下面要解锁了，对方处理器还得等待呢
         */
        if (data_flags & CSD_FLAG_LOCK)
            csd_unlock(data);
    }
}

static void csd_unlock(struct call_single_data *data)
{   …
    //清楚 CSD_FLAG_LOCK，释放对方处理器
    data->flags &= ~CSD_FLAG_LOCK;
}
```

第 3 章　调度与实时性

3.1　Tick

Tick 是内核的生命脉搏。对于 Fair 调度类和 RT 调度类中的 SCHED_RR 线程，Tick 的每次到来将意味调度实际的临近。Tick 还意味当前 User 进程 Signal 的处理时机。对于处于 Sleep 状态处理器，在满足定时 Timer 的基础上，则需消除不必要的 Tick。

3.1.1　Local timer

Local timer 可以叫做 Ticker，是 Tick 之源，是处理器局部的。其产生的中断 Tick 只被对应处理器处理。

在 ARM CA9 架构下有两种方式的 Local timer。

（1）SPI 类型的 Local timer。将中断分发给对应的处理器即可，但是这样似乎有些别扭。

（2）PPI 类型的 Local timer。只被局部处理器看到，不占用 SPI 号，自然和谐。

CA9 早期的 SOC 实现主要是 SPI 类型的 Local timer，在 CA9 的后期版本的 SOC 实现中，以 PPI 类型 Local timer 为主流。

Local timer 的具体实现与处理器相关，以 Exynos4412 为例。内核里每颗处理器对应的 Local timer 都对应一个 struct mct_clock_event_device 的实例：

```
struct mct_clock_event_device mct_tick[NR_CPUS];
```

每颗处理器的 Local timer 的初始化如下：

```
static void exynos4_mct_tick_init(struct clock_event_device *evt)
{
    //取出 CPU 号
    unsigned int cpu = smp_processor_id();
    //用 CPU 号索引对应数组
    mct_tick[cpu].evt = evt;
    /*对于每个处理器，其 Local timer 控制寄存器的基地址都不同，偏移量为 0x100：
    #define EXYNOS4_MCT_L_BASE(x)        (_EXYNOS4_MCT_L_BASE + (0x100 * x))
    而且其中断类型为 PPI，由此可以推断，Exynos4412 里面每颗处理器都有一个 Local timer
    的具体实现，而且在其事件发生时仅产生 PPI 中断。这是较为优美的 Local timer 实现，
    比 4 个 SPI 中断好多了*/
    mct_tick[cpu].base = EXYNOS4_MCT_L_BASE(cpu);
```

```
    evt->name = mct_tick[cpu].name;
    evt->cpumask = cpumask_of(cpu);
    // set_next_event set_mode 函数设置
    evt->set_next_event = exynos4_tick_set_next_event;
    evt->set_mode = exynos4_tick_set_mode;
    //该 Local timer 的能力
    evt->features = CLOCK_EVT_FEAT_PERIODIC | CLOCK_EVT_FEAT_ONESHOT;
    //优先级
    evt->rating = 450;
    …
    //向系统中注册该 timer
    clockevents_register_device(evt);
    exynos4_mct_write(TICK_BASE_CNT,        mct_tick[cpu].base        +
    MCT_L_TCNTB_OFFSET);
    // mct_int_type 的值是 MCT_INT_PPI，不用考虑 SPI 中断了
    if (mct_int_type == MCT_INT_SPI) {
        if (cpu == 0) {
            …
        } else {
            mct_tick1_event_irq.dev_id = &mct_tick[cpu];
            //挂载中断
            setup_irq(IRQ_MCT_L1, &mct_tick1_event_irq);
            //中断定向
            irq_set_affinity(IRQ_MCT_L1, cpumask_of(1));
        }
    } else {
        //这里使能该 PPI 中断，在初始化刚完成时 GIC 的 PPI 中断是 disable 的
        gic_enable_ppi(IRQ_PPI_MCT_L);
    }
}
```

3.1.2　Tick 挂载

在 CA9 SMP 架构下，每个 Core 使用自己的 Local timer 作为自己的 tick 之源。

在 arch/arm/kernel/smp.c 中，内核为每颗处理器定义了自己的 struct clock_event_device：

```
static DEFINE_PER_CPU(struct clock_event_device, percpu_clockevent);
```

在每个 Core 初始化时，需要把自己的 Tick 源打开：

```
void __cpuinit percpu_timer_setup(void)
{
    /*取得自己的 CPU 号*/
    unsigned int cpu = smp_processor_id();
    /*通过自己的 CPU 号索引自己的 struct clock_event_device*/
```

```
    struct clock_event_device *evt = &per_cpu(percpu_clockevent, cpu);

    evt->cpumask = cpumask_of(cpu);
    evt->broadcast = smp_timer_broadcast;
    /*SOC 架构相关的 Local timer 初始化，每种 SOC 的将自己的 Local timer 参数填入
    struct clock_event_device结构,然后调用void clockevents_register_device(…),
    向内核注册这个 tick 设备*/
    if(local_timer_setup(evt))
        broadcast_timer_setup(evt);
}
```

接下来 Core 检查该 struct clock_event_device 是否符合自己的 Tick 设备标准。

```
static int tick_check_new_device(struct clock_event_device *newdev)
{
    …

    cpu = smp_processor_id();
    /*检查该 struct clock_event_device 指示的 cpu mask 里有没有自己，要是没有自己，
    说明不符合，直接退出*/
    if (!cpumask_test_cpu(cpu, newdev->cpumask))
        goto out_bc;

/*取出该 Core 当前使用的 struct tick_device*/
    td = &per_cpu(tick_cpu_device, cpu);
    curdev = td->evtdev;

    /* 检查该 struct clock_event_device 是否是当前 Core 的 Local 设备*/
    if (!cpumask_equal(newdev->cpumask, cpumask_of(cpu))) {

        /*该 struct clock_event_device 不是当前 Core 的 Local 设备 */
        /*尝试将该设备的中断定向分发给当前 Core，Tick 需要的就是中断，如果不是 Local
        timer，但是如果其中断能定向分发过来，也可以*/
        if (!irq_can_set_affinity(newdev->irq))
            goto out_bc;

        /*当前的 struct clock_event_device 是 Local 设备,就没有必要再换成非 Local
        设备了，即使精度再高也没必要，这种 Local 设备在架构上更和谐 */
        if (curdev && cpumask_equal(curdev->cpumask, cpumask_of(cpu)))
            goto out_bc;
    }

    /*在新老设备中间选择 */
    if (curdev) {
        /*更喜欢 CLOCK_EVT_FEAT_ONESHOT 的设备，宁可在每次 Tick 时重新设置，也不要
        周期性的，ONESHOT 的好处在于可以在 idle 时做 tickless 。一般的 Local timer
```

```
        都是两者属性兼具的：CLOCK_EVT_FEAT_PERIODIC | CLOCK_EVT_FEAT_ONESHOT*/
        if ((curdev->features & CLOCK_EVT_FEAT_ONESHOT) &&
            !(newdev->features & CLOCK_EVT_FEAT_ONESHOT))
            goto out_bc;
        /*别的因素都考虑了，以优先级做评判标准 */
        if (curdev->rating >= newdev->rating)
            goto out_bc;
    }
    …
   /*新的 struct clock_event_device 已选定，下面将挂载它*/
    tick_setup_device(td, newdev, cpu, cpumask_of(cpu));
    …
}

/*Tick 设备的建立 */
static void tick_setup_device(struct tick_device *td,
                    struct clock_event_device *newdev, int cpu,
                    const struct cpumask *cpumask)
{
    ktime_t next_event;
    /*记录当前 Core 的 Tick 中断函数*/
    void (*handler)(struct clock_event_device *) = NULL;

    if (!td->evtdev) {
        …
        /*这时 Tick 还没有启震起来，还不知道下一次 Tick 的时间，所以做周期性震荡
         */
        td->mode = TICKDEV_MODE_PERIODIC;
    } else {
    /* Tick 已经启震起来，知道下一次 Tick 的时间，struct tick_device 的 mode 保持
    原来的方式，并记录 Tick 的中断 handler */
        handler = td->evtdev->event_handler;
        next_event = td->evtdev->next_event;
        td->evtdev->event_handler = clockevents_handle_noop;
    }
    /*切换 struct clock_event_device*/
    td->evtdev = newdev;

    /*如果不是 Local 设备，将其中断分发设置为定向到该 Core */

    if (!cpumask_equal(newdev->cpumask, cpumask))
        irq_set_affinity(newdev->irq, cpumask);

    …
/* 下 一 步 ， 如 果 是 第 一 次 挂 载  td->mode  ==  TICKDEV_MODE_PERIODIC ， struct
```

```
clock_event_device 的成员变量 void (*event_handler)(…); 将 被 设 置 为 void
tick_handle_periodic(struct clock_event_device *dev)，这是该 Core 的 Tick 中
断函数。
如果是更新挂载，则将该 struct clock_event_device 的工作方式设置为单发模式：
CLOCK_EVT_MODE_ONESHOT，并重新对硬件编程设置下一发的时间点*/

    if (td->mode == TICKDEV_MODE_PERIODIC)
        tick_setup_periodic(newdev, 0);
    else
        tick_setup_oneshot(newdev, handler, next_event);
}
```

以上均没考虑 tick broadcast，在绝大多数手机里这个功能是 disable。

3.1.3　Tick 产生

作为系统脉搏，Tick 发生时，内核还会产生一些动作，如编程下一次 Tick 中断，更新时间，触发 Timer 软中断等动作，而最关键的是将 Tick 送入调度器。

```
/*Tick 处理的框架函数，一方面将 Tick 送给内核，一方面编程时钟寄存器预定下一次 Tick 的到
来时间*/
Void tick_handle_periodic(struct clock_event_device *dev)
{
    int cpu = smp_processor_id();
    ktime_t next;
    /*这里将 Tick 送进 scheduler，Time 和 Timer 维护的工作也在这里*/
    tick_periodic(cpu);

    /*下面要确定下一次 Tick 的时间，如果该设备不支持 CLOCK_EVT_MODE_ONESHOT，那么
    返回即可，接受周期中断*/
    if (dev->mode != CLOCK_EVT_MODE_ONESHOT)
        return;
    /*对下一次 Tick 中断产生的时间进行硬件编程 */
    next = ktime_add(dev->next_event, tick_period);
    for (;;) {
        if (!clockevents_program_event(dev, next, ktime_get()))
            return;
        …
    }
}
//维护时间，并将 Tick 继续送给内核其他组件
static void tick_periodic(int cpu)
{
/*每个 Core 都有自己 Timer 队列，但是 Time 只能由一个 Core 来维护*/
    if (tick_do_timer_cpu == cpu) {
```

```
        …
        /*Time 维护，记下 jiffies*/
        do_timer(1);
        …
    }
/*Timer 及 Tick 维护*/
    update_process_times(user_mode(get_irq_regs()));
  …
}
//将 Tick 送给调度器，维护处理器自己的 Timer 队列、检查 posix timer
void update_process_times(int user_tick)
{
    /*timer 维护*/
    run_local_timers();
    /*rcu 维护*/
    rcu_check_callbacks(cpu, user_tick);
    …
    /*把 tick 送进调度器*/
    scheduler_tick();
    /*posix timer 维护*/
    run_posix_cpu_timers(p);
}
```

3.2　Fair 调度类

在 2.6 内核的后期，引入了调度类。调度类与调度队列密不可分。调度队列是实体，每颗处理器一个，而调度类为策略。主要的调度类有 RT、Fair、Idle 三种。而对于每颗处理器，所有这些调度类所管辖的线程队列都被组织在该处理器的 struct rq 队列中。

3.2.1　Fair 调度类的负载均衡

负载均衡操作的第一个任务是统计出在一个调度域最繁忙的处理器负载。这分为两步，第一步在调度域内找出最繁忙的 struct sched_group，第二步在该 struct sched_group 中找出最繁忙的处理器。对于 CA9 架构，尽管每颗处理器都有自己的 struct sched_domain 结构，但是这些 struct sched_domain 的跨度范围是包含在线的所有处理器，所以，逻辑上看，这些处理器都处在一个调度域中。再者，对于 CA9 架构，每个 struct sched_group 只包含一个处理器，这些 struct sched_group 又被组织成一个环形链表（参见调度域构建部分）。所以在 CA9 架构上，最繁忙的处理器寻找可以看作遍历该 struct sched_group 环形链表，寻找负载最重的处理器。

```
/*该函数找出在一个调度域内负载最高的 struct sched_group。参数 struct sd_lb_stats
*sds 用于记载统计结果。struct sched_domain *sd 为当前执行负载均衡操作处理器对应
```

```
struct sched_domain 结构
*/
static inline void update_sd_lb_stats(struct sched_domain *sd, int this_cpu,
            enum cpu_idle_type idle, const struct cpumask *cpus,
            int *balance, struct sd_lb_stats *sds)
{
    struct sched_domain *child = sd->child;
    /*当前处理器对应的 struct sched_group，该函数的主题是顺着这个 struct
    sched_group 遍历该调度域覆盖的所有 struct sched_group*/
    struct sched_group *sg = sd->groups;
    //用于存放该 struct sched_group 负载的统计结果
    struct sg_lb_stats sgs;
    int load_idx, prefer_sibling = 0;
    …
    do {
        int local_group;
        /*struct sched_group 的成员变量 unsigned long cpumask[0];指出了该
        struct sched_group 覆盖的处理器，对于 CA9 架构，其指出了包含处理器的对应位，
        在 I.mx6Q 架构上，其取值为 1、2、4、8。这里检测要统计的 struct sched_group
        是否就是当前处理器对应的 struct sched_group*/
        local_group = cpumask_test_cpu(this_cpu, sched_group_cpus(sg));
        memset(&sgs, 0, sizeof(sgs));
        //提取该 struct sched_group 负载信息到 sgs 中
        update_sg_lb_stats(sd, sg, this_cpu, idle, load_idx,
                local_group, cpus, balance, &sgs);
        /*当前处理器对应 struct sched_group 无法做 loadbalance 的目标，无事可做返
        回*/
        if (local_group && !(*balance))
            return;

        …

        if (local_group) {
            //统计目标处理器的负载
            sds->this_load = sgs.avg_load;
            …
        } else if (update_sd_pick_busiest(sd, sds, sg, &sgs, this_cpu)) {
            //将新统计处理出来的 sgs 与原有 sgs 中最忙碌的比较，选出繁忙处理器
            sds->max_load = sgs.avg_load;
            sds->busiest = sg;
            …
        }
        …
        //链表里的下一个 struct sched_group
        sg = sg->next;
```

```
    } while (sg != sd->groups);
}

/*统计一个 struct sched_group 内的所有处理器负载，对于 CA9 架构，每个 struct
sched_group 内置有一颗处理器*/
static inline void update_sg_lb_stats(struct sched_domain *sd,
            struct sched_group *group, int this_cpu,
            enum cpu_idle_type idle, int load_idx,
            int local_group, const struct cpumask *cpus,
            int *balance, struct sg_lb_stats *sgs)
{
    …

   /*若需要统计的 struct sched_group 包含当前处理器，balance_cpu 记录下该 struct
    sched_group 里第一颗处理器，对于 CA9 即为当前处理器*/
    if (local_group)
        balance_cpu = group_first_cpu(group);

    /*针对 struct sched_group 里的每一颗处理器操作*/
    for_each_cpu_and(i, sched_group_cpus(group), cpus) {
        struct rq *rq = cpu_rq(i);

        if (local_group) {
        /* 若是当前处理器（目标处理器）的 struct sched_group，即把负载往该处理器上
        拉*/
            if (idle_cpu(i) && !first_idle_cpu) {
                //Tick 发生在 Idle，cpu 醒来试图分担别的处理器的负载
                first_idle_cpu = 1;
                balance_cpu = i;
            }
            /*取出该处理器的负载，对于 Fair 调度类，在线程进队出队时都是调整处理器的
            负载，参见相关章节*/
            load = target_load(i, load_idx);
        } else {
         /*从当前处理器角度看别的处理器，取出其负载*/
            load = source_load(i, load_idx);
            …
        }
        /*累计 struct sched_group 内所有处理器负载，对于 CA9 这里只做一次*/
        sgs->group_load += load;
        …
    }

    /*
     在 schedule 中若某个运行队列可运行为 0，面临 Idle 时，出现以下情况
```

```
     */
    if (idle != CPU_NEWLY_IDLE && local_group) {
        /*这种情况要做 balance 的处理器为 struct sched_group 里第一颗处理器，否则
        无法做 balance。这种处理器似乎不是物理处理器，应该是逻辑处理器，多颗逻辑处理
        器共享相同的计算单元，SMT 似乎符合这种情况。但对于 CA9 物理处理器就是逻辑处理
        器，不会出现该情况*/
        if (balance_cpu != this_cpu) {
            *balance = 0;
            return;
        }
        …
    }

    …
}
```

事实上，实际情况要复杂得多，可能根本就没有符合条件的最繁忙处理器，或者当前处理器不能满足作为负载转移目标条件。这里还隐藏着另外一个影响负载的因素，即 Fair 调度类和 Rt 调度类是分别统计负载的，Fair 的负载值并没有直接记录 RT 调度类的线程情况。但是如果某个调度队列上 RT 负载较重，则影响到其 Fair 调度类复制值的消减，这样运行 RT 线程的处理器上，其 Fair 调度类的负载值就消逝较慢，使其逐渐成为 Fair 调度类的 busiest。

负载均衡时机为定时、idle、newly_idle、nohz_idle_balance 四种情况，前三种情况的特点是目标处理器都是当前处理器。而尽管 nohz_idle_balance 导致的 loadbalance 框架与前三者相同，但其目标处理器却不是当前处理器。nohz_idle_balance 基本机理是：nohz_idle 在 idle 状态去除了禁止了 tick，所以在处理器进入 nohz idle 之间，就把自己标志在 idle_cpus_mask 位图中。在别的 active 处理器发生 tick 时，将该处理器作为 loadbalance 目标处理器。

接下来就可以分析负载均衡的框架函数了：在选出最繁忙的处理器之后，接着就可以做负载均衡了——把最繁忙处理器上的线程拉到目标处理器上（本节提到当前处理器都是描述前三种调度时机）。

```
/*
负载均衡框架函数，前 4 个参数都是描述当前处理器的，最后一个参数描述负载均衡完成的结果
*/
static int load_balance(int this_cpu, struct rq *this_rq,
            struct sched_domain *sd, enum cpu_idle_type idle,
            int *balance)
{
    int ld_moved, all_pinned = 0, active_balance = 0;
    …

redo:
  /*找出最繁忙的 struct sched_group*/
```

```
    group = find_busiest_group(sd, this_cpu, &imbalance, idle,
                cpus, balance);
    //如果没有满足条件的 struct sched_group，直接返回
    if (*balance == 0)
        goto out_balanced;

    /*在找到的 struct sched_group 里面找到繁忙的逻辑处理器，对于 CA9 这是一对一的关
    系*/
    busiest = find_busiest_queue(sd, group, idle, imbalance, cpus);

    //如果没有满足条件的逻辑处理器，直接返回
    if (!busiest) {
        schedstat_inc(sd, lb_nobusyq[idle]);
        goto out_balanced;
    }

    // ld_moved 记录下有多少线程发生了处理器间移动
    ld_moved = 0;
    if (busiest->nr_running > 1) {
        all_pinned = 1;
        local_irq_save(flags);
        double_rq_lock(this_rq, busiest);
        /*把最繁忙处理器上的线程移动到当前处理器上，摘取出原有队列再入队的过程*/
        ld_moved = move_tasks(this_rq, this_cpu, busiest,
                    imbalance, sd, idle, &all_pinned);
        double_rq_unlock(this_rq, busiest);
        local_irq_restore(flags);

        /*
        若目标处理器不是当前处理器，置位 TIF_NEED_RESCHED 使得目标处理器有机会运行
        新线程，这是 nohz_idle_balance 发生的情况
        */
        if (ld_moved && this_cpu != smp_processor_id())
            resched_cpu(this_cpu);

        /* 最繁忙处理器上的线程都是处理器绑定的，不能被转移到别的处理器上 */
        if (unlikely(all_pinned)) {
            cpumask_clear_cpu(cpu_of(busiest), cpus);
            if (!cpumask_empty(cpus))
                goto redo;
            goto out_balanced;
        }
    }
    …
}
```

3.2.2 Fair 调度类的处理器选择

Fair 调度类的目标是尽量让所有线程得到公平的运行时机，无论对于手机还是嵌入式系统，绝大多数线程都是属于 Fair 调度队列的。

```
/*
 该函数作为 struct sched_class fair_sched_class 变量的 int  (*select_task_rq)
(…); 函数，在当唤醒 Fair 调度类的线程时，为线程选择合适的处理器
*/
static int
select_task_rq_fair(struct task_struct *p, int sd_flag, int wake_flags)
{
    struct sched_domain *tmp, *affine_sd = NULL, *sd = NULL;
    int cpu = smp_processor_id();
    //取出待唤醒线程 p 之前运行的处理器
    int prev_cpu = task_cpu(p);
    int new_cpu = cpu;
    int want_affine = 0;
    int want_sd = 1;
    int sync = wake_flags & WF_SYNC;
   //在 CA9 的架构下该条件成立
    if (sd_flag & SD_BALANCE_WAKE) {
      //检查线程 p 能否运行在当前处理器上
        if (cpumask_test_cpu(cpu, &p->cpus_allowed))
            want_affine = 1;
        new_cpu = prev_cpu;
    }

    rcu_read_lock();
    for_each_domain(cpu, tmp) {
        …

        /*
        若线程 p 的先前处理器与当前处理同属于一个调度域，且线程 p 又能运行在当前处理器
        上。则 affine_sd 置位当前处理器对应 struct sched_domain。这种情况是 CA9 处
        理器常见路径
        */
        if (want_affine && (tmp->flags & SD_WAKE_AFFINE) &&
            cpumask_test_cpu(prev_cpu, sched_domain_span(tmp))) {
            affine_sd = tmp;
            want_affine = 0;
        }

        …
```

```
        }

        if (affine_sd) {
            …
            /*若当前 cpu 处于 idle 状态则选择当前 cpu，否则若线程 p 先前 cpu 处于 idle 状态则选择其先前 cpu，否则在该调度域覆盖范围内的处理器选择一个处于 idle 状态的运行 p。若找不到 idle 状态处理器则直接使用 prev_cpu。这是 CA9 架构最常见路径*/
            new_cpu = select_idle_sibling(p, prev_cpu);
            goto unlock;
        }

        /*否则在调度域覆盖处理器里选择一个最空闲的处理器来运行 p*。该路径不常见，发生情况为当前处理器不在线程 p 的可运行处理器列表中*/
        while (sd) {
            //寻找最空闲 struct sched_group
            group = find_idlest_group(sd, p, cpu, load_idx);
            …
            //在最空闲 struct sched_group 中寻找最空闲处理器
            new_cpu = find_idlest_cpu(group, p, cpu);
            …
            }
        }
        …
        return new_cpu;
    }
```

3.3　RT 调度类

3.3.1　RT 调度类的基本结构

调度策略属于 SCHED_FIFO、SCHED_RR 的 task_struct，属于 RT 调度类，RT 调度类的队列组织结构以优先级为核心。

（1）处在运行状态的每个相同优先级的 task_struct 被串在同一队列。

（2）内核构造一个以长为最大优先级的链表头数组，上述 task_struct 队列以优先级为索引挂在对应的链表头上。

（3）通过一个长为最大优先级的位图表示对应优先级是否有处在运行状态的 task_struct。

定义如下所示：

```
struct rt_prio_array {
    //表示对应优先级是否有可运行 task_struct
    DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1); /* include 1 bit for delimiter
*/
```

```
    //挂载 task_struct 队列的链表头
    struct list_head queue[MAX_RT_PRIO];
};
```

调度类提供了入队函数的主要策略就是依据其优先级索引对应链表头：

```
static void __enqueue_rt_entity(struct sched_rt_entity *rt_se, bool head)
{
    …
    //根据优先级索引链表数组
    struct list_head *queue = array->queue + rt_se_prio(rt_se);
…
    //将其挂入该链表头
    if (head)
        list_add(&rt_se->run_list, queue);
    else
        list_add_tail(&rt_se->run_list, queue);
    /*以优先级为索引置位运行时位图，表示当前优先级链表上有处于运行状态的
    task_struct*/
    __set_bit(rt_se_prio(rt_se), array->bitmap);
    //当前运行队列可运行 task_struct 数目增一
    inc_rt_tasks(rt_se, rt_rq);
}

/*暴露给调度器的入队函数，rq 为将要加入的调度队列，p 即为需要加入的 task_struct*/
static void enqueue_task_rt(struct rq *rq, struct task_struct *p, int flags)
{
    …
    //挂入调度队列
    enqueue_rt_entity(rt_se, flags & ENQUEUE_HEAD);
  /*若该 task_struct 不是调度队列 rq 的当前线程，且该 task_struct 可以在多个处理器上
  运行，则将其该队列的 pushable_tasks 链表，在该 CPU 负载严重时，该 pushable_tasks
  链表上的 task_struct 将会被推送到别的处理器上*/
    if (!task_current(rq, p) && p->rt.nr_cpus_allowed > 1)
        enqueue_pushable_task(rq, p);
}
```

RT 调度类的 Tick 函数充分解释了 SCHED_FIFO、SCHED_RR 调度策略的不同。

```
static void task_tick_rt(struct rq *rq, struct task_struct *p, int queued)
{
    …
    /*对于 SCHED_FIFO 策略的 task_struct，Tick 无效，这种类型的如果不是自己主动放弃
    CPU，不被更高优先级的 SCHED_FIFO 策略 task 抢占，就不眠不休一直跑下去*/
    if (p->policy != SCHED_RR)
        return;
    /*对于 SCHED_RR 调度策略的 task 仍有时间片的概念，Tick 到来其 time_slice 仍然被
```

```
    递减*/
    if (--p->rt.time_slice)
        return;
    //递减到 0，恢复时间片的默认值
    p->rt.time_slice = DEF_TIMESLICE;

    /*
    把当前 SCHED_RR 调度策略的 task 从调度队列里摘掉，并将其挂到队尾。这个操作体现了轮
    转。在 RT 里只有一个 task 时，即使 SCHED_RR 调度策略的 task 也能享受到 FIFO 的待遇
    */
    if (p->rt.run_list.prev != p->rt.run_list.next) {
        requeue_task_rt(rq, p, 0);
        set_tsk_need_resched(p);
    }
}

/*调度器使用的下一运行 task 选择函数，该函数蕴含的一个重要机制是，如果当前调度类选择不
出来合适的 task，则调度器会在下一个调度类里寻找，直到 idle task*/

static struct task_struct *_pick_next_task_rt(struct rq *rq)
{
    struct sched_rt_entity *rt_se;
    struct task_struct *p;
    struct rt_rq *rt_rq;

    rt_rq = &rq->rt;

    //如果 RT 调度类没有处在运行态的 task，则说明没有合适的 task
    if (unlikely(!rt_rq->rt_nr_running))
        return NULL;
    //如果 RT 调度类到了带宽限制阀值，则说明没有合适的 task
    if (rt_rq_throttled(rt_rq))
        return NULL;
    //选取下一最高优先级的 task，参考进队操作
    do {
        rt_se = pick_next_rt_entity(rq, rt_rq);
        …
    } while (rt_rq);
    //索引到新 task
    p = rt_task_of(rt_se);
    //在新 task 被选取时与 RT 调度队列的 clock_task 时钟同步
    p->se.exec_start = rq->clock_task;

    return p;
}
```

常规应用通常是不会跑到 RT 调度队列中的，只有在某些特定实时应用中才能大量使用。而对于 Android 系统，早期的版本 RT 队列基本没有用到，到了后期版本进行了优化，将 surfacefilnger 之类的线程进行实时性改造放入了 RT 队列。

3.3.2 Rt_Bandwidth

为了限制系统中 RT 调度类过度占用 CPU，内核提供了 rt_bandwidth 机制，即设置一个百分比，当 RT 运行队列的 task 占据 CPU 的时间超过了这个阈值，则调度器将 RT 调度类视而不见，将剩下的时间分配给其他调度类的 task。

尽管该机制通过全局变量 int sysctl_sched_rt_runtime 来控制是否开启。但是在大多数 ARM 处理器厂商提供的 BSP 里都是默认开启的，所以对于某些依靠最高优先级的 FIFO 实时线程来说要注意，这种情况下有些时间是不受控制的。若默认最高优先级且处于运行态的 FIFO 时，线程能 totally 控制系统将存在潜在 bug。

```
/*
RT 调度类的 shchedule_tick 里会调用该函数
 */
static void update_curr_rt(struct rq *rq)
{
    struct task_struct *curr = rq->curr;
    struct sched_rt_entity *rt_se = &curr->rt;
    struct rt_rq *rt_rq = rt_rq_of_se(rt_se);
    u64 delta_exec;

    if (curr->sched_class != &rt_sched_class)
        return;
/*在当前 task 被调度时及 tick 发生时，其 se.exec_start 与 RQ 的 clock_task 同步，参见上文。而在每次 tick 时，RQ 的 clock_task 与系统时间同步。void scheduler_tick(void)-> static void update_rq_clock(struct rq *rq)-> static void update_rq_clock_task(struct rq *rq, s64 delta)。所以这里求出来的差值之和是 RT 调度类运行的时间长度*/
    delta_exec = rq->clock_task - curr->se.exec_start;
    …
    //当前 task 的 exec_start 时间与运行队列时间同步
    curr->se.exec_start = rq->clock_task;
    …
    //如果禁用了 rt_bandwidth 直接返回即可
    if (!rt_bandwidth_enabled())
        return;

    for_each_sched_rt_entity(rt_se) {
        …
        if (sched_rt_runtime(rt_rq) != RUNTIME_INF) {
```

```
            //防抢占防中断
            raw_spin_lock(&rt_rq->rt_runtime_lock);
            //rt_time 记录了 RT 调度类运行的时间长度
            rt_rq->rt_time += delta_exec;
            /*查看当前 RT 运行时间长度是否超过了 rt_bandwidth 机制的限制，如果超过
            了限制，则进入调度器*/
            if (sched_rt_runtime_exceeded(rt_rq))
            /*如果为真，当前 task 置为 TIF_NEED_RESCHED 运行状态，即使运行态的 FIFO
            task 也得交出处理器*/
                resched_task(curr);
            raw_spin_unlock(&rt_rq->rt_runtime_lock);
        }
    }
}

//检查否超过了 rt_bandwidth 机制的限制
static int sched_rt_runtime_exceeded(struct rt_rq *rt_rq)
{
    //runtime 取至 RQ 的 rt_runtime 成员变量，这是限制的时间点
    u64 runtime = sched_rt_runtime(rt_rq);
    …
    //如果实际运行时间超过了限制的运行时间，则 rq 的成员变量 rt_throttled 为真
    if (rt_rq->rt_time > runtime) {
        rt_rq->rt_throttled = 1;
        …
    }

    return 0;
}
```

这之后的机制是，当前 CPU 进入调度器，调度器再次重选线程，当前 RT 调度队列被 throttled，所以 RT task 选择返回 NULL，调度器将继续尝试 Fair 直到 Idle。这其中蕴藏着一个问题是，如果 Fair 里没有 task，该处理器将会进入到 Idle 中，尽管这时 RT 里还有 task 没有运行完毕，是在所有队列中都没有 task 才 Idle，还是不管如何都从 RT 里剥夺一定的运行时间，是一个值得讨论的问题。

为了控制 RT 调度队列的 throttled 与 active，内核使用一个 hrtimer——sched_rt_period_timer 来处理这个机制。

```
//Hrtimer 时钟的处理函数
static enum hrtimer_restart sched_rt_period_timer(struct hrtimer *timer)
{
    …

    for (;;) {
       //取出当前时间
        now = hrtimer_cb_get_time(timer);
```

```
        /*forward 该 hrtimer， rt_b->rt_period 的一个周期默认为一秒，
rt_b->rt_runtime 为 0.95 秒。这些值在 void init_rt_bandwidth(...)里被初始化*/
        overrun = hrtimer_forward(timer, now, rt_b->rt_period);
        if (!overrun)
            break;
        /*实际工作函数，在一个周期的边沿，检查是否重启 RT 调度队列，或者当前 RT 调度队
        列已无可运行 task，则返回 idle*/
        idle = do_sched_rt_period_timer(rt_b, overrun);
    }
    //是否 idle 来决定该 Hrtime 是否重新 fire
    return idle ? HRTIMER_NORESTART : HRTIMER_RESTART;
}

static int do_sched_rt_period_timer(struct rt_bandwidth *rt_b, int overrun)
{
    //idle 默认为 1
    int i, idle = 1;
    …
    //取出在线处理器位图
    span = sched_rt_period_mask();
    //只针对在线处理器操作
    for_each_cpu(i, span) {
        int enqueue = 0;
        struct rt_rq *rt_rq = sched_rt_period_rt_rq(rt_b, i);
        struct rq *rq = rq_of_rt_rq(rt_rq);

        raw_spin_lock(&rq->lock);
        //检查该运行队列是否有实际运行时间记录
        if (rt_rq->rt_time) {
            u64 runtime;

            raw_spin_lock(&rt_rq->rt_runtime_lock);
            //有实际运行时间记录且被 rt_throttled
            if (rt_rq->rt_throttled)
                balance_runtime(rt_rq);
            //限制的时间长度
            runtime = rt_rq->rt_runtime;
            //通常 overrun 为 0，这里 rt_rq->rt_time 被清零
            rt_rq->rt_time -= min(rt_rq->rt_time, overrun*runtime);
            //rt_rq->rt_time 被清零，且 rt_rq->rt_throttled 为真
            if (rt_rq->rt_throttled && rt_rq->rt_time < runtime) {
                // rt_throttled 状态被清零
                rt_rq->rt_throttled = 0;
                enqueue = 1;

                /*
                发生在 RT 队列 throttled 时，Fair 里却没有处在运行状态的 task
                 */
                if (rt_rq->rt_nr_running && rq->curr == rq->idle)
```

```
                    rq->skip_clock_update = -1;
                }
                //发生在 RT 队列运行完毕以后 hrtimer 才到来
                if (rt_rq->rt_time || rt_rq->rt_nr_running)
                    idle = 0;
                raw_spin_unlock(&rt_rq->rt_runtime_lock);
            } else if (rt_rq->rt_nr_running) {
              /*该 RT 队列对应的 CPU 刚被唤醒，还没来得及执行第一个 RT task，或者第一个 RT
              task 还没执行到第一个 TICK*/
                idle = 0;
                if (!rt_rq_throttled(rt_rq))
                    enqueue = 1;
            }
            /* enqueue 为真导致该 CPU 进调度器,(对于最后一种情况导致多进入一次 schedule)
            新的周期开始*/
            if (enqueue)
                sched_rt_rq_enqueue(rt_rq);
            raw_spin_unlock(&rq->lock);
        }
        /*idle 为 1 的原因在于：系统压根就没有 RT task 或者 RT 队列的 task 在一个周期的限
        制之内就运行完毕了*/
        return idle;
    }
```

3.3.3　负载均衡与抢占

负载均衡用在 RT 调度类上似乎不太合适，虽然存在推拉动作，但这并不是为了平衡处理器的负载，而是以满足实时性为第一要务前提下的处理器间任务分配。

1. 在唤醒时刻

（1）在处理器选择时刻：若被唤醒线程优先级低于当前处理器当前线程优先级且当前处理器当前线程不能在别的处理器上运行，则为被唤醒线程选择一个运行低优先级线程的处理器（仍有可能再次选择到当前处理器，参见唤醒部分）。

（2）在将被唤醒线程挂入目标处理器运行队列时刻：

① 若被唤醒线程优先级在目标 RT 运行队列 struct rt_rq 中最高，则在 struct highest_prio 的结构里记录下被唤醒线程。

② 若被唤醒线程可以在多个处理器上运行，则将其挂入目标 RT 运行队列 struct rt_rq 的 struct plist_head pushable_tasks;链表。值得一提的是，task 挂入该链表的顺序是以其优先级为顺序，这样其被 push 走的时候也是优先级由高而低依次进行。而每次 push 线程的时候是在其余处理器上线程最高优先级低于被 push 的线程才发生 push 动作，否者该线程还是不会被 push 走。

（3）在优先级检查时刻。

① 检查目标 RT 运行队列上的当前线程是否可以抢占，如果被唤醒线程优先级较高，

则若目标 RT 运行队列上的当前线程将被标志 TIF_NEED_RESCHED，出现被唤醒线程与目标 RT 运行队列上的当前线程优先级相同又出现如下情况。

- 若被唤醒线程能在多个处理器上运行，则什么都不做，因为被唤醒线程现在处于目标 RT 运行队列 struct rt_rq 的 struct plist_head pushable_tasks;链表中，它有可能被推出去或者拉出去。
- 若目标 RT 运行队列的当前线程和被唤醒线程都只能在目标处理器上运行，什么也做不了。
- 若目标 RT 运行队列的当前线程可在其他处理器上运行，则被唤醒线程抢占当前线程，这之后被抢占的线程将有机会被推出去或者拉出去到别的处理器。

② 在唤醒操作的最后一动作 task_woken 时刻。

若被唤醒线程不能抢占当前运行队列的当前线程，且被唤醒线程可以在多个处理器上运行，且目标 RT 运行队列 struct rt_rq 的 struct plist_head pushable_tasks;链表不为空，在对于该链表执行 push 操作。

2. 在调度器的 pre_schedule 时刻

这时处理器还未进行新的线程遴选，但是下一个被切换上的线程的优先级低于当前线程的优先级将发生位操作。这里的逻辑是当前线程无法运行，且不是被高优先级线程抢占，那么可以从别的处理器上拉过来优先级较高的线程。

3. 在调度器的 post_schedule 时刻

切换完毕，大位已定，没选上的线程将被 push 到别的处理器。

3.3.4 基础操作

针对多处理的优先级管理，Linux 内核使用了如下结构：

```
struct cpupri {
    /*该数组以优先级为索引，每一数据项表示对应优先级的处理器位图。若某处理器对应位在该
    位图为真，则表示该处理器运行的线程的最高优先级即为该数据项的索引*/
     struct cpupri_vec pri_to_cpu[CPUPRI_NR_PRIORITIES];
     //以处理器号为索引，数据项的值即为该处理器运行的线程的最高优先级
     int             cpu_to_pri[NR_CPUS];
};

//在每次将新的线程加入 RT 调度队列时都要更新该位图
void cpupri_set(struct cpupri *cp, int cpu, int newpri)
{
     int *currpri = &cp->cpu_to_pri[cpu];
     //取出 cpu 上原有的最高优先级
     int oldpri = *currpri;
     int do_mb = 0;

     //struct cpupri 优先级数值不同于线程优先级数值，值越大优先级越高
```

```
        newpri = convert_prio(newpri);

        /*若新挂入的线程优先级，与该处理器运行队列中最高优先级的线程同级，则无需进一步操作，
        返回*/
        if (newpri == oldpri)
            return;
        if (likely(newpri != CPUPRI_INVALID)) {
            //以优先级索引处理器位图
            struct cpupri_vec *vec = &cp->pri_to_cpu[newpri];
            //置位处理器对应位
            cpumask_set_cpu(cpu, vec->mask);
            …
        }
        if (likely(oldpri != CPUPRI_INVALID)) {
            struct cpupri_vec *vec  = &cp->pri_to_cpu[oldpri];
          …
          //清除掉处理器原有对应位
            cpumask_clear_cpu(cpu, vec->mask);
        }
        //记录下该处理器上的最高优先级
        *currpri = newpri;
    }
```

在唤醒线程和 push 操作时往往需要寻找另外一颗处理器，这时的目标是其上运行的线程优先级最低，这样 push 和 wake up 的线程才有机会立刻运行。

```
    //其中 p 为被唤醒或者被 push 的线程
    int cpupri_find(struct cpupri *cp, struct task_struct *p,
            struct cpumask *lowest_mask)
    {
        int idx = 0;
        //算出 p 的优先级在 struct cpupri 的对应值
        int task_pri = convert_prio(p->prio);
        …
        /*从优先级最低往上索引，目标是找优先级最低的，且其优先级不能高于 p 的优先级，否则
        即使把 push 或者 wakeup 上去也无法获得处理器*/
        for (idx = 0; idx < task_pri; idx++) {
            struct cpupri_vec *vec  = &cp->pri_to_cpu[idx];
            …
            /*该处理器可运行处理位于与上优先级 idx 对应位图，所得值才是 p 的目标处理器*/
            if (lowest_mask) {
                cpumask_and(lowest_mask, &p->cpus_allowed, vec->mask);
                /*计算结果位于 lowest_mask，若没有合适的处理器再次寻找下一个优先级对应
                cpu 位图*/
                if (cpumask_any(lowest_mask) >= nr_cpu_ids)
                    continue;
            }
            //找到了合适的处理器
            return 1;
        }
```

```
    /*所有优先级低于 p 的位图都找了，没有发现合乎条件的，返回 0，说明没找到*/
    return 0;
}
```

3.4 调 度 器

3.4.1 调度域的构建

从 Linux 2.6 内核的后期版本开始，内核使用了调度域、调度组等概念来组织处理器之间的关系。由于笔者目前手头上能找到的核心数目最多的 ARM 开发板仅有 I.MX6Q，本文仅分析 4 核 CA9 架构下的调度域结构。

首先针对不同架构的层次关系内核使用如下结构来完成调度域的初始化：

```
static struct sched_domain_topology_level default_topology[] = {
//超线程处理器
#ifdef CONFIG_SCHED_SMT
    { sd_init_SIBLING, cpu_smt_mask, },
#endif
    …
//普通处理器，对于 CA9，仅有该项
    { sd_init_CPU, cpu_cpu_mask, },
//NUMA 机器
#ifdef CONFIG_NUMA
    { sd_init_NODE, cpu_node_mask, SDTL_OVERLAP, },
    …
#endif
    { NULL, },
};
```

该数组反映了处理器逻辑的拓扑关系，若处理器逻辑单元有不同的层次，则在每个层次上都有对应拓扑描述结构 struct sched_domain_topology_level。其中的 sched_domain_mask_f mask;成员函数描述了在该层次逻辑处理器对应分组关系。对于 CA9 处理器，这个结构数组退化为仅有一项。其处理器分组即为所有在线处理器。

以 struct sched_domain_topology_level 为基础，调度域实现函数如下：

```
static int build_sched_domains(const struct cpumask *cpu_map,
                  struct sched_domain_attr *attr)
{   …
    /* 针对每一颗在线处理器*/
    for_each_cpu(i, cpu_map) {
        struct sched_domain_topology_level *tl;
        sd = NULL;
        /*在 sched_domain_topology 的每个层次构建调度域，对于 CA9 架构
        sched_domain_topology 退化为仅有一项，所以整个函数退化为针对每个处理
        器构建其调度域*/
        for (tl = sched_domain_topology; tl->init; tl++) {
```

```
            /*对每个处理器构建其调度域，对于 CA9 架构，sd 永远为 0，参见下文*/
            sd = build_sched_domain(tl, &d, cpu_map, attr, sd, i);
            …
        }
        /*对于 CA9 架构 sd->child 为 0，只有一层，但是对于其他架构，这就构成了
        调度域的父子关系*/
        while (sd->child)
            sd = sd->child;
        //记录下处理器的调度域
        *per_cpu_ptr(d.sd, i) = sd;
    }

    /*为每个调度域构建 struct sched_group*/
    for_each_cpu(i, cpu_map) {
        /*对于 CA9 架构，其调度域仅为一层，这里就退化为针对每颗处理器的 struct
        sched_domain 构建其 struct sched_group*/
        for (sd = *per_cpu_ptr(d.sd, i); sd; sd = sd->parent) {
            //计算一下该调度域覆盖多少处理器
            sd->span_weight = cpumask_weight(sched_domain_span(sd));
            if (sd->flags & SD_OVERLAP) {
                /*对于 CA9 架构，其调度域属性里没有 SD_OVERLAP，不可能走
                到这里*/
                if (build_overlap_sched_groups(sd, i))
                    goto error;
            } else {
                //构建 struct sched_group，参见下文
                if (build_sched_groups(sd, i))
                    goto error;
            }
        }
    }

/*执行到这里，对于 I.MX6Q 处理器，在其 4 颗处理器均 online 情况下，调度域、调度组的构建
如图 3-1 所示*/
    …
    rcu_read_lock();
   //每颗处理器的运行对于 struct rq 与其调度域 struct sched_domain 绑定起来
    for_each_cpu(i, cpu_map) {
        sd = *per_cpu_ptr(d.sd, i);
        cpu_attach_domain(sd, d.rd, i);
    }
    rcu_read_unlock();

    …
    return ret;
}
```

最终得到如图 3-1 所示的调度域结构。

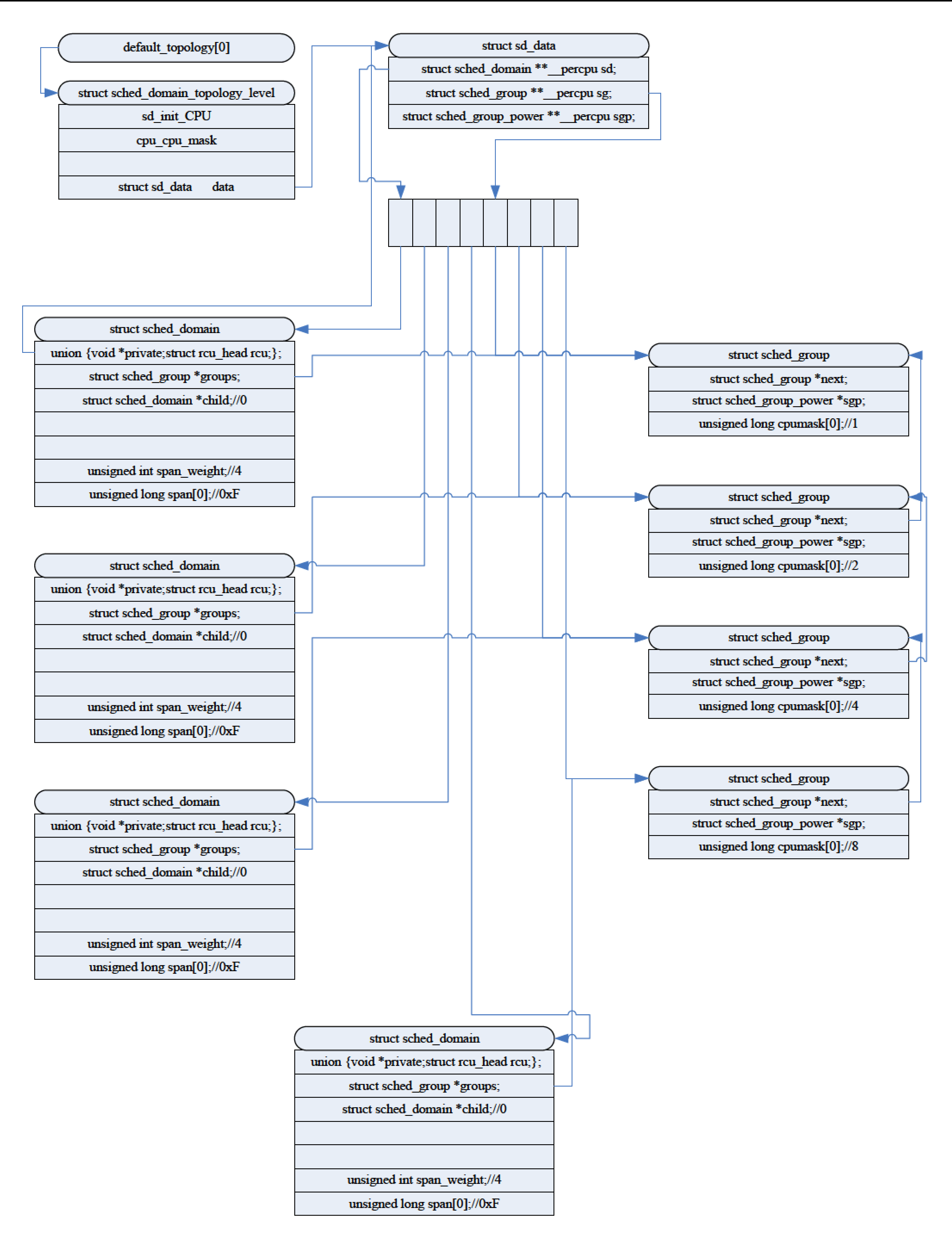


图 3-1 调度域和调度组的构成

```
/*单个调度域的构建。对于 CA9，tl 即为{ sd_init_CPU, cpu_cpu_mask, },, child 为 0*/
struct sched_domain *build_sched_domain(struct sched_domain_topology_
level *tl,
        struct s_data *d, const struct cpumask *cpu_map,
        struct sched_domain_attr *attr, struct sched_domain *child,
        int cpu)
```

```
{
    /*struct sched_domain 结构本身的内存已经分配，这里通过 percpu 的指针找到对应
    cpu 的 struct sched_domain，并将其赋值为 SD_CPU_INIT*/
    struct sched_domain *sd = tl->init(tl, cpu);
    if (!sd)
        return child;
    /*给调度域的 unsigned long span[0];赋值，该值描述了调度域覆盖的处理器范围，对
    于 CA9，其值为 cpu_online_mask*/
    cpumask_and(sched_domain_span(sd), cpu_map, tl->mask(cpu));
    /*child 参数反映出处理器的逻辑关系，从最底层的 smt 开始，层次逐渐增长，且当前调度
    域是前一级调度与的父亲。对于 CA9 架构不用考虑这个*/
    if (child) {
        sd->level = child->level + 1;
        sched_domain_level_max = max(sched_domain_level_max, sd->level);
        child->parent = sd;
    }
    //CA9 的调度域 child 和 parent 都为 0
    sd->child = child;
    set_domain_attribute(sd, attr);

    return sd;
}

/*
 * build_sched_groups will build a circular linked list of the groups
 * covered by the given span, and will set each group's ->cpumask correctly,
 * and ->cpu_power to 0.
 代码注释很好地解释了该函数的作用，值得一提的是针对调度域覆盖范围内的所有 struct
sched_group 构建，仅在第一颗处理器 struct sched_group 构建时就全部完成
 */
static int
build_sched_groups(struct sched_domain *sd, int cpu)
{
    …

    //若该处理器不是其调度域覆盖范围的第一颗处理器，返回
    if (cpu != cpumask_first(sched_domain_span(sd)))
        return 0;
    …
    //针对调度域覆盖范围的每一颗处理器
    for_each_cpu(i, span) {
        struct sched_group *sg;
        /*其 struct sched_group 内存早已分配，这里在 struct sched_group *sg 里
        记录其地址*/
        int group = get_group(i, sdd, &sg);
```

```
        …
        /*清楚 struct sched_group 的 unsigned long cpumask[0];*/
        cpumask_clear(sched_group_cpus(sg));
        sg->sgp->power = 0;
        /*对于 CA9，该循环的逻辑关系得到简化，即为针对 i 对应的 struct sched_group，
        将其 unsigned long cpumask[0];对应的第 i 位置 1*/
        for_each_cpu(j, span) {
            if (get_group(j, sdd, NULL) != group)
                continue;

            cpumask_set_cpu(j, covered);
            cpumask_set_cpu(j, sched_group_cpus(sg));
        }
        /*将该调度域覆盖范围内的所有 struct sched_group 串起来*/
        if (!first)
            first = sg;
        if (last)
            last->next = sg;
        last = sg;
    }
    last->next = first;

    return 0;
}
```

3.4.2 调度器

在基于 Linux 的系统中，无论内核态还是用户态，任何时刻、任何一行代码都必定属于一个线程。线程运行时的边界就是调度器。调度工作由调度算法和切换组成，两者之间泾渭分明。调度算法是指在调度队列选择线程的策略，调度算法没有好坏，只有适合与不适合之分。

切换的作用是保存被调度下去线程的 Context 且恢复下一个占用处理器的线程 Context 的工作。ARM 架构处理器大量使用到了协处理器，而协处理器有着大量的寄存器 Context，这些 Context 的保存恢复将很大程度影响到切换时间。但是系统中使用到协处理器的线程并不多，如 Android 系统中仅仅自编解码多媒体线程才会用到 Neon，而且它们同时出现在同一个处理器核心运行队列的概率又更小了。所以这就为切换优化提供了前提，ARM 切换器的做法，若下一线程不使用协处理器则协处理器的 Context 可不必切换。

切换时系统中值得关注的另外一个问题是 Cache，若 Cache 是依据于虚拟地址组织的，那么若发生跨进程线程的切换时势必要更新 Cache，否则数据就混乱了。ARM 早期架构如 ARM9 的 Cache 组织无疑是 VIVT 的，这样对于 Linux 这种有着独立虚拟地址空间的进程模型每次进程切换 Cache 都要更新一遍。这里不得不承认早期 WINCE5 之前的多个进程共享 4G 虚拟地址空间的方式在这个方面更胜一筹。在 CA8 之后的 ARM 实现中 L2 和 L1 Data

Cache 通常实现为 PIPT，只有 L1 Instruction Cache 才采用 VIPT，这样在进程切换时虚拟地址空间的问题仅影响到 L1 Instruction Cache。

```
/*
调度器函数，线程边界，该函数有两个潜在的参数：当前线程，当前处理器调度队列
 */
static void __sched __schedule(void)
{   …
need_resched:
    //禁止抢占
    preempt_disable();
    //获取当前处理器号
    cpu = smp_processor_id();
    //根据处理器号索引该处理器对应的运行队列
    rq = cpu_rq(cpu);
    //rcu 相关，参见 rcu 部分
    rcu_note_context_switch(cpu);
    //当前 cpu 运行队列成员变量 struct task_struct *curr，指向当前线程
    prev = rq->curr;
    …
    //关中断
    raw_spin_lock_irq(&rq->lock);

    switch_count = &prev->nivcsw;
    /*处理当前线程在非被抢占且退出运行态时的情况*/
    if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
       //若当前线程
        if (unlikely(signal_pending_state(prev->state, prev))) {
            prev->state = TASK_RUNNING;
        } else {
        /*普通情况，当前线程在自身遇到无法继续运行的情况，而导致线程切换，将当前线
        程从运行队列中取出*/
            deactivate_task(rq, prev, DEQUEUE_SLEEP);
            /*struct task_struct 结构的成员变量 int on_rq;指出当前线程已不在运行
            队列*/
            prev->on_rq = 0;

            if (prev->flags & PF_WQ_WORKER) {
                struct task_struct *to_wakeup;

                to_wakeup = wq_worker_sleeping(prev, cpu);
                if (to_wakeup)
                    try_to_wake_up_local(to_wakeup);
            }
        }
```

```
        switch_count = &prev->nvcsw;
    }

    pre_schedule(rq, prev);
    /*若当前处理器的运行队列上没有处于运行态的线程，需要从别的处理器上拉些线程到自己的
    运行队列*/
    if (unlikely(!rq->nr_running))
        idle_balance(cpu, rq);
    /*处于运行态的线程，并没有从运行队列中被拿掉，处理非运行态线程也不能再放入运行队列，
    对不同的调度类有不同的处理，RT 调度类做的工作是整理运行队列及当前线程的运行时间，
    且对运行态的 task 重新决定其运行的处理器*/
    put_prev_task(rq, prev);
    /*选取下一个合适的线程，对于每颗处理器，其选择新线程的顺序依次是 RT 调度类、Fair
    调度类和 Idle 调度类。若选择了 Idle 调度类这说明该处理器要进入 Idle*/
    next = pick_next_task(rq);
    //清楚当前线程的需要调度标志
    clear_tsk_need_resched(prev);
    rq->skip_clock_update = 0;
    /*再次选取的线程，可能就是当前线程，这种情况直接调出调度函数即可，否则需要进行切换
    操作*/
    if (likely(prev != next)) {
        rq->nr_switches++;
        //当前运行队列更新自己的当前线程指针
        rq->curr = next;
        ++*switch_count;
        //线程切换，从这里开始当期线程发生改变
        context_switch(rq, prev, next); /* unlocks the rq */
        /*新老线程都从这里出来，只不过老线程跑到这里时是其被下一次调度选中，甚至是在
        另外一颗处理器上，新线程是完成了虚拟内存切换（新进程），将上次保存在寄存器信息
        恢复到处理器后即刻运行到这里*/
        cpu = smp_processor_id();
        rq = cpu_rq(cpu);
    } else
        raw_spin_unlock_irq(&rq->lock);

    post_schedule(rq);

    //清除当前线程的禁止抢占标识
    preempt_enable_no_resched();
    /*在新线程完成切换到运行到这里的时候发生了中断，且唤醒了更高优先级的线程，导致当前
    线程被抢占（新线程在 void finish_task_switch(..)时会打开中断）*/
    if (need_resched())
        goto need_resched;
}
```

3.5 唤　　醒

3.5.1 唤醒与抢占

唤醒意味着将线程状态切换成可运行，本节首先分析内核的唤醒操作，在 SMP 架构处理器环境下，除了涉及基本运行队列的进队操作，还需要通过相应调度类进行处理器的选择。

1. 唤醒

```
/*该函数作为等待队列唤醒的缺省函数，是唤醒指定线程最常用函数*/
static int
try_to_wake_up(struct task_struct *p, unsigned int state, int wake_flags)
{
    …
    //锁 raw_spinlock_t pi_lock;关中断
    raw_spin_lock_irqsave(&p->pi_lock, flags);
    …
    cpu = task_cpu(p);
    /*这里的发生情况为： p 在另外一颗处理器上已置为不可运行状态，接下来进入调度器，但
    是在另外一颗处理器上调度器还没走到 raw_spin_lock_irq(&rq->lock);，这时在当前
    处理器就唤醒了该 p，这时 p->on_rq 仍有效，当前处理器与另一颗处理器竞争对方处理器
    的 raw_spinlock_t lock;（函数 static int ttwu_remote(…) 通过 rq =
    __task_rq_lock(p);来竞争对方 raw_spinlock_t lock;)，若当前处理器获胜，该 p
    通过快捷操作被重新置位运行状态，若对方处理器获胜该 p，wake up 操作进入下面常规唤
    醒操作*/
    if (p->on_rq && ttwu_remote(p, wake_flags))
        goto stat;

#ifdef CONFIG_SMP
    /*  调度器在完成切换时会将切换上的处理器 struct task_struct 中的 int on_cpu;
    置 1，而将被切换掉的处理器 struct task_struct 中的 int on_cpu;置零。若某架构处
    理器切换时希望能同时打开中断，则宏 __ARCH_WANT_INTERRUPTS_ON_CTXSW 和
    __ARCH_WANT_UNLOCKED_CTXSW 被定义。这意味着在切换操作的进行中，该处理器的中断
    是打开的且其切换函数不持有其运行的队列 struct rq 的 raw_spinlock_t lock;锁*/
    while (p->on_cpu) {
#ifdef __ARCH_WANT_INTERRUPTS_ON_CTXSW
        /*
    这里发生以下情况:
     (1) 当前处理器上唤醒该 struct task_struct *p，但该 p 在对方处理器上正处在已经
从 rq 的运行队列被取掉，其 on_rq 失效，且在被切换掉的过程中，其 on_cpu 依然有效。这时，
当前处理器只需在获得对方处理器 rq 的 raw_spinlock_t lock;后将该 p 重新挂入其运行队列
即可。
```

```
　　（2）当前处理器上唤醒该 struct task_struct *p，同时对方处理器其在 p 切换过程中
发生的了中断，且该中断做唤醒 p 的操作。这时当前处理器持有 p 的 raw_spinlock_t pi_lock;，
而对方处理器持有却在该 p 的 raw_spinlock_t pi_lock;上等待，但对方处理器隐性持有
on_cpu 为真的条件。所以接下来当前处理器在该 p 重新挂入其运行队列后即刻释放掉该 p 的
raw_spinlock_t pi_lock;
          */
        if (ttwu_activate_remote(p, wake_flags))
            goto stat;
#else
        cpu_relax();
#endif
    }
    /*
     接下来是常规的唤醒操作
     */
    smp_rmb();

    p->sched_contributes_to_load = !!task_contributes_to_load(p);
    p->state = TASK_WAKING;
    /*对于 fair 调度类，为 p 选择一颗负载最低的处理器，对于 RT 调度类选择处理器的原则是
    以保证该 p 能及时运行为前提，分为以下情况：
     （1）若 p 上次运行的处理器上当前运行的不是 RT 线程，则继续选择 p 上次运行的处理器，
在后面的 wake up 操作中 p 将抢占该处理器。
     （2）若 p 上次运行的处理器上当前运行的是 RT 线程，且该 RT 线程优先级高于 p 或者该 RT
线程绑定了该处理器，则需要为 p 选择另外一颗处理器其运行
*/
    if (p->sched_class->task_waking)
        p->sched_class->task_waking(p);

    //为该 p 选择一颗合适的处理器
    cpu = select_task_rq(p, SD_BALANCE_WAKE, wake_flags);
    if (task_cpu(p) != cpu) {
        wake_flags |= WF_MIGRATED;
        //为 p 设置新的处理器
        set_task_cpu(p, cpu);
    }
#endif /* CONFIG_SMP */
/*若该 p 选定的处理器为当前处理器，把 p 挂到对应处理器调度队列中，或者将 p 挂到对方处理器
的 struct task_struct *wake_list;队列中，并向对方处理器发送处理器间中断。对方处理
器在收到中断后处理该 struct task_struct *wake_list;队列。参见下文*/
    ttwu_queue(p, cpu);
    …
out:
    //解 raw_spinlock_t pi_lock;开中断
    raw_spin_unlock_irqrestore(&p->pi_lock, flags);
```

```
    return success;
}
```

2. 抢占

唤醒在逻辑上属于调度的一部分，是调度时机产生的源泉。在新的线程被唤醒时，内核将检查当前线程是否能够抢占其对应运行队列上的当前线程。在线程被唤醒后内核使用 static void check_preempt_curr(…)来检查抢占条件。

```
static void check_preempt_curr(struct rq *rq, struct task_struct *p, int flags)
{
    const struct sched_class *class;
    if (p->sched_class == rq->curr->sched_class) {
    /*被唤醒 p 所属调度类与指定 rq 当前线程属于同一调度类，则将抢占条件的检查交与具体调度类*/
        rq->curr->sched_class->check_preempt_curr(rq, p, flags);
    } else {
    /*对与被唤醒 p 所属调度类与指定 rq 当前线程不属于同一调度类的情况，其原则是 RT 调度类抢占 Fair 调度类，从最高优先级的 RT 调度类往下检查*/
        for_each_class(class) {
            //指定 rq 当前线程所属调度类优先级更高，没有必要抢占
            if (class == rq->curr->sched_class)
                break;
            //线程 p 所属调度类优先级更高，抢占
            if (class == p->sched_class) {
                resched_task(rq->curr);
                break;
            }
        }
    }

    …
}
```

3.5.2　跨处理器分发线程

当一颗处理器将线程均衡给别的处理器时或唤醒了一个应该在其他处理上运行的线程时，这些线程并不是立即处于目标处理器的运行队列，而是位于其 wake_list，这时需要该处理器发起处理器间中断，以通知目标处理器接受这些线程。

```
//首先，发送方处理器发起 RESCHEDULE
void smp_send_reschedule(int cpu)
{   //处理器间中断，cpu 为对方处理器号
    smp_cross_call(cpumask_of(cpu), IPI_RESCHEDULE);
}
```

```
/*接着，在目标侧的处理器上将接收到 IPI 中断 IPI_RESCHEDULE，从而激活其线程队列*/
asmlinkage void __exception_irq_entry do_IPI(int ipinr, struct pt_regs
*regs)
{
    …
    switch (ipinr) {
    …
    //处理器线程队列激活分支，在这里响应对方处理器
    case IPI_RESCHEDULE:
        scheduler_ipi();
        break;
     …
    }
    …
}
```

接着目标侧处理中依次检查自己的 wake_list，将其中线程一一激活，该 wake_list 往往是对方处理器因为自身繁忙或者由于优先级原因挂载过来的线程队列。

```
//list 即为该处理运行队列的 wake_list
static void sched_ttwu_do_pending(struct task_struct *list)
{
    …
   //使用 ttwu_do_activate 函数依次执行激活操作
    while (list) {
        struct task_struct *p = list;
        list = list->wake_entry;
        ttwu_do_activate(rq, p, 0);
    }
…
}
```

3.5.3 抢占

抢占分为用户态抢占和内核态抢占。前者是 Linux 基本特性，后者在打开内核配置开关时被激活。抢占是实时性的基础，但是实时性的好坏决定于内核访问控制的粒度。通过将内核访问控制更细力度的修改，可以获得更好的实时性。本节仅分析单核处理器下抢占情况。

1. 避免抢占的方法

当内核有段代码段需要受到保护而不能受高优先级 task 打扰时，为了保护这段代码段，内核在其前后加入：

```
preempt_disable();
```

受保护的代码段：

```
preempt_enable();
```

其中 preempt_disable()往该 task 的 struct thread_info 的 preempt_count 里 0～7 位加 1，表示自己已经进入非抢占代码段。

2. 禁止抢占状态时发生了中断，唤醒高优先级线程，试图抢占

这时一个硬件中断进入，导致另外一个 task 被唤醒，而这个 task 优先级高于当前优先级，并且当前 task 的调度类属于 RT，这时当前 task 的 flags 的重新调度位被置位。接着该中断返回时检查当前 task 的 preempt_count 非零，所以不能被抢占，硬件中断只好乖乖的退出，同时恢复当前 task 的受保护代码段的执行。

然后当当前 task 的受保护代码段执行完毕，它调用了 preempt_enable()，将自己在 preempt_count 里的 0～7 位所加的 1 清除。但是这时内核又检查自己当前 task 的 flags 的 TIF_NEED_RESCHED 是否被置位。很明显刚才进入硬件中断，将该位置位，于是内核开始抢占，当前 task 被设置为 PREEMPT_ACTIVE 状态，内核进入 schedule 选择新的 task。

接下来分析上述场景对应的代码：

中断处理函数使用 static int try_to_wake_up(…)唤醒一个 task 时，这时调度队列的状态发生了变化，有必要检查一下是否将当前 task 抢占掉。

```
static int try_to_wake_up(struct task_struct *p, unsigned int state, int
sync)
{
    …
    struct rq *rq;
    …
    //rq 是当前处理器运行队列
    rq = task_rq_lock(p, &flags);
    …
    //p 是要被抢占的 task
    check_preempt_curr(rq, p, sync);
    …
}

static inline void check_preempt_curr(struct rq *rq, struct task_struct *p,
int sync)
{
//找到当前运行队列的当前运行 task，再找到这个当前 task 的对应调度类
    rq->curr->sched_class->check_preempt_curr(rq, p, sync);
}
```

不同于早期的 Linux 2.6 内核，那时只要配置了内核抢占，一旦唤醒了一个高优先级的 task，当前 task 不由分说直接被拖下处理器，现在的内核将这个工作交给调度类。

```
static inline void check_preempt_curr(struct rq *rq, struct task_struct *p,
int sync)
{
    rq->curr->sched_class->check_preempt_curr(rq, p, sync); //由调度类来决定
}

//先看 RT 调度类，事实上 RT 调度类与 Linux 2.6 早期的调度策略很相似
static void check_preempt_curr_rt(struct rq *rq, struct task_struct *p, int
sync)
{
/*被抢占 task 如果是 rt task，处理比较干脆利落，管你抢占 task 属于什么调度类，只看你的
优先级，如果被唤醒 task 小于当前 task 的 prio（prio 越小优先级越大）*/
    if (p->prio < rq->curr->prio) {
//告诉内核当前 task 需要拉下处理器
        resched_task(rq->curr);
        return;
    }
    …
}
/*再看 Fair 调度类的情况*/
static void check_preempt_wakeup(struct rq *rq, struct task_struct *p, int
sync)
{
    …
    // rt task 要来抢占 cfs 里的当前 task
    if (unlikely(rt_prio(p->prio))) {
        resched_task(curr);
        return;
    }

    /*到了这里说明抢占 task p 不属于 RT 调度类，正常情况下 task p 就应该属于 Fair 调度
    类了。如果是 Idle 调度类的 task，显然当前内核实现不允许其抢占 Fair 调度类。但是如
    果又写了一个调度类，而这个 task p 就属于这个新的调度类呢？所以如果要再加一个新的调
    度类，而且希望能够抢占 Fair 里的 task，就需要改动这里了*/
    if (unlikely(p->sched_class != &fair_sched_class))
        return;
    //到了这里无论是抢占 task 还是被抢占 task 都属于 Fair 了
    …
    /*如果当前 task 重新调度位已经被置位，那就不用管了，返回。其实这给了在 Fair 调度类
    task 被抢占的一个灵活机制，在这种情形下，如果确认这个 task 的确需要被换下，那么在
    中断处理函数中直接将其置位好了，Fair 调度类本身会依从这个策略*/
    if (test_tsk_need_resched(curr))
        return;
    /*如果抢占 task 的调度策略不是 NORMAL，不用理，直接返回*/
    if (unlikely(p->policy != SCHED_NORMAL))
```

```
        return;
    //如果当前 task 的调度类是 IDLE，换掉它
    if (unlikely(curr->policy == SCHED_IDLE)) {
        resched_task(curr);
        return;
    }
    /*OK，既然内核的调度策略就不允许这种 WAKEUP 时的抢占，那就没什么事了，返回*/
    if (!sched_feat(WAKEUP_PREEMPT))
        return;

    /*内核的调度策略允许这种 WAKEUP，当前内核状态满足重新调度条件，置位*/
    if (sched_feat(WAKEUP_OVERLAP) && (sync ||
            (se->avg_overlap < sysctl_sched_migration_cost &&
             pse->avg_overlap < sysctl_sched_migration_cost))) {
        resched_task(curr);
        return;
    }
    find_matching_se(&se, &pse);
    /*根据自己调度策略，而不是仅仅比较 prio 大小来决定抢占 task 是否更需要占有 cpu*/
    if (wakeup_preempt_entity(se, pse) == 1)
        resched_task(curr);
}
```

再看 Idle 调度类，这个比较简单，这里最好说话，直接退让。

```
static void check_preempt_curr_idle(struct rq *rq, struct task_struct *p,
int sync)
{
    resched_task(rq->idle);
}
```

需要说明的是，在这些调度类的 static void check_preempt_curr(...)函数里，并没有直接切换 task，这个时候还在中断处理函数中，而且保护的代码段还没执行完。所以这时仅仅是将当前 task 的重新调度位置位。

```
static void resched_task(struct task_struct *p)
{
        set_tsk_need_resched(p); //置位 task 的 flags 重新调度位

}
```

到了这里中断处理程序完成了自己的工作，也唤醒了需要唤醒 task，到了中断处理程序退出的时候。如下是内核发生中断时的代码分析：

```
__irq_svc:
    svc_entry
#ifdef CONFIG_PREEMPT       //内核被配置为可抢占
```

```
    get_thread_info tsk      //找到当前 task 的 struct thread_info 指针
    ldr r8, [tsk, #TI_PREEMPT]  //把 preempt_count 放到寄存器 r8 里
    add r7, r8, #1           @ increment it //中断本身也是不能被抢占的
    str r7, [tsk, #TI_PREEMPT]
#endif
    irq_handler         //中断处理函数，唤醒另一个 task，当前置位需要调度
    //中断从这里退出
#ifdef CONFIG_PREEMPT
    //恢复中断前的当前 task 的 preempt_count
    str r8, [tsk, #TI_PREEMPT]      @ restore preempt count
    //取出当前 task 的 TI_FLAGS
    ldr r0, [tsk, #TI_FLAGS]        @ get flags
    //先看一下当前 task 的 preempt_count 是否为 0
    teq r8, #0               @ if preempt count != 0
    /*如果 preempt_count 不为 0,r0 被置 0,说明无论如何不能被抢占,如果 preempt_count
    不为 0, r0 就是 task 的 TI_FLAGS, 如果被置位了_TIF_NEED_RESCHED 将会导致抢占发
    生。代码转向 svc_preempt，所以这就是为什么在代码段中加上 preempt_disable()和
    preempt_enable()能起到保护不被抢占的原因*/
    movne   r0, #0              @ force flags to 0
    tst r0, #_TIF_NEED_RESCHED
    blne    svc_preempt
#endif
    ldr r0, [sp, #S_PSR]        @ irqs are already disabled
    msr spsr_cxsf, r0
    …
    //恢复之前或的状态
    ldmia   sp, {r0 - pc}^          @ load r0 - pc, cpsr
 UNWIND(.fnend      )
ENDPROC(__irq_svc)
```

这里，中断返回了，受保护的代码继续运行，当受保护的代码运行完遇到了 preempt_enable()。这里抢占真实发生了，另外一个 task 被推上处理器。

```
//开抢占
#define preempt_enable() \
do { \
    preempt_enable_no_resched(); \ //这里将自己加在 preempt_count 的 1 减掉
    barrier(); \   //内存屏蔽
    preempt_check_resched(); \
} while (0)

//调度时机检测
#define preempt_check_resched() \
do { \
    if (unlikely(test_thread_flag(TIF_NEED_RESCHED))) \ 看一下自己是否需要重
    新调度
```

```
        preempt_schedule(); \   //抢占
} while (0)
//抢占调度
asmlinkage void __sched preempt_schedule(void)
{
    //取出 struct thread_info 指针
    struct thread_info *ti = current_thread_info();
    /*检查 preempt_count 和中断是否被禁止。这两个情况任何一个发生都不能抢占。在
    Linux 2.6 后期的内核里，处理器中断被禁止也是不可抢占状态*/
    if (likely(ti->preempt_count || irqs_disabled()))
        return;
    do {
        //将自己置位被抢占了
        add_preempt_count(PREEMPT_ACTIVE);
        /*进入调度器，调度器会选择比自己更适合 task 来运行，当前 task 被拉下处理器*/
        schedule();
        /*当被抢占掉 task 又获得了运行机会，从这里重新开始*/
        sub_preempt_count(PREEMPT_ACTIVE);
    …
    } while (need_resched());
}
```

3. 普通的内核态抢占

需要禁止抢占的内核代码毕竟是少数，大部分情况下，内核是没有禁止抢占的，这时与上文不同的是在中断返回时，走到了 svc_preempt。

```
svc_preempt:
//lr 里保存的就是上一条指令 blne svc_preempt 的下一个地址
    mov r8, lr
//跳到 asmlinkage void __sched preempt_schedule_irq(…)
1:  bl  preempt_schedule_irq         @ irq en/disable is done inside
/*又一次获得运行机会，再次检查是否需要调度，如果需要，还是要让出 CPU。从获得运行机会到
这里，可能新的中断又发生了。所以要再次检查*/
    ldr r0, [tsk, #TI_FLAGS]         @ get new tasks TI_FLAGS
    tst r0, #_TIF_NEED_RESCHED
//返回
    moveq   pc, r8               @ go again
    b   1b

asmlinkage void __sched preempt_schedule_irq(void)
{    //取出当前 task 的 struct thread_info
    struct thread_info *ti = current_thread_info();

    do {
        //标志自己被抢占
```

```
            add_preempt_count(PREEMPT_ACTIVE);
            //打开中断
            local_irq_enable();
            //进入调度器切换
            schedule();
            //关闭中断
            local_irq_disable();
    //清除自己的被抢占位
            sub_preempt_count(PREEMPT_ACTIVE);
            barrier();
        } while (need_resched());
    }
```

值得一提的是，在这种状态下被抢占 task 换下 CPU 时，其内核栈存放着中断到来时自己的 context，所以再次获得 CPU 时，要从先前中断进来的地方退出去，这样才能退到自己原来的执行地址。

第 4 章　Signal

Signal 是系统里的一种特殊机制，是指线程在从内核态切换到用户态时，脱离原有用户态执行路径而进入预先设置的 Signal Handler 的机制。Signal Handler 的前提发生在中断、地址转换异常和系统调用向用户态的返回，系统不能确定其准确的执行时刻。因为一般应用获得处理器的时间是不能被控制的，所以尽管 Tick 中断和定时器能够准时到来并触发当前线程的 Signal Handler，但是内核并不能保证应用线程获得处理器，所以时钟中断并不能保证命中靶心，更何况其他的随机中断了。

Signal 处理与处理器架构相关，本章内容包含了 IWMMX 协处理器的处理分析，IWMMX 的处理方式也适用于 NEON 协处理器。

4.1　信 号 发 送

信号发送是指当系统中某个事件产生时，需要某个线程的某个 Signal Handler 做出响应，这时内核就在这个线程的 thread_info 里做一个记号，待时机合适时执行，而对于一些严重的事件，不需要用户态 Signal Handler 的响应，内核直接就可以对响应的线程做出处理。

```
//信号发送函数
static int __send_signal(int sig, struct siginfo *info, struct task_struct *t,
            int group, int from_ancestor_ns)
{
    …

  /*检查该信号是否被指定的 struct task_structblock 或 ignore*/
   if (!prepare_signal(sig, t, from_ancestor_ns))
       return 0;

  /*group 指出了该信号是被送往进程（线程组）还是被送往单独的线程*/
   pending = group ? &t->signal->shared_pending : &t->pending;

   /* 如果该信号类型为小于 SIGRTMIN 非实时信号，且指定接收对象已经接收到了该信号，则
      直接跳过信号挂载操作*/
   if (legacy_queue(pending, sig))
       return 0;

   /*参数 info 携带的信息 SEND_SIG_FORCED 指出这是发自内核的诸如 SIGUSR1、SIGSTOP
```

```
        之类的信号，也无需信号挂载操作*/
    if (info == SEND_SIG_FORCED)
        goto out_set;
    …

    //分配该信号的 struct sigqueue 结构，以携带该信号其他信息
    q = __sigqueue_alloc(sig, t, GFP_ATOMIC | __GFP_NOTRACK_FALSE_POSITIVE,
        override_rlimit);
    if (q) {
       //挂入线程组或线程的 peding 队列
        list_add_tail(&q->list, &pending->list);
        switch ((unsigned long) info) {
        case (unsigned long) SEND_SIG_NOINFO:
        …
        default:
            //复制携带的信息
            copy_siginfo(&q->info, info);
            if (from_ancestor_ns)
                q->info.si_pid = 0;
            break;
        }
    } else if (!is_si_special(info)) {
        …
    }

out_set:
    …
  //设置信号集对应位
    sigaddset(&pending->signal, sig);
   /*评估最终接受信号的线程、设置信号 pending 标志、唤醒接收信号线程等操作。参见下文*/
    complete_signal(sig, t, group);
    return 0;
}


static void complete_signal(int sig, struct task_struct *p, int group)
{
    struct signal_struct *signal = p->signal;
    struct task_struct *t;

    /*
    评价指定的接收线程是不是该信号的合适目标线程，其评价标准如下：如果信号被指定线程
    block，或者指定线程正在结束生命，或者指定线程被 debug，或者指定线程当前不占用处理
    器，或者指定线程有 pending 没被处理的 signal，那么指定线程都不是该信号合适的接收
    体。这里避开指定线程当前不占用处理器情况的原因在于普通信号的执行时机必定是从内核态
```

```
向用户态返回的路上，如果不是当前线程其信号执行必定会遭到延迟，所以尽量选择占用处理器的线程。而避开有 pending  signal 的指定线程则是为了使信号执行更加均匀，不至于都堆在一个线程上
 */
if (wants_signal(sig, p))
    t = p;
else if (!group || thread_group_empty(p))
    /*若要寻找另外合适的线程，则对象的目标必须是发往线程组，如果不是线程组，直接
      返回，因为对于单一线程的进程。若其占用处理器且已经置位了 TIF_SIGPENDING
      就不需要唤醒该线程了*/
    return;
else {
    /*
    搜寻线程组里的线程，找到合适线程
     */
    t = signal->curr_target;
    while (!wants_signal(sig, t)) {
        t = next_thread(t);
        if (t == signal->curr_target)
            /*遍寻线程组，找到的线程已经被其他信号选择过了，这里不需要进一步对其
              做 TIF_SIGPENDING 置位或者唤醒操作了，因为上一次的信号发送过程
              已经做过了*/
            return;
    }
   //找到合适的线程
    signal->curr_target = t;
}


if (sig_fatal(p, sig) &&
    !(signal->flags & (SIGNAL_UNKILLABLE | SIGNAL_GROUP_EXIT)) &&
    !sigismember(&t->real_blocked, sig) &&
    (sig == SIGKILL ||
     !tracehook_consider_fatal_signal(t, sig))) {

    if (!sig_kernel_coredump(sig)) {
        /*SIGQUIT、SIGILL、SIGTRAP、SIGABRT、SIGBUS 等信号发生，杀死整个线程组*/
     ….
        return;
    }
}

/*
对指定线程置位 TIF_SIGPENDING 唤醒 TASK_INTERRUPTIBLE 睡眠状态的指定线程
```

```
         */
        signal_wake_up(t, sig == SIGKILL);
        return;
    }
```

4.2 信号执行

如上文所述，Signal 执行时机是其从内核返回用户态时。在该 task 的返回途径上，内核会检查是否有 pending 的 signal 没有处理，如果有 signal pending，则执行信号处理。内核为 Signal 处理函数执行需要做如下几项工作。

（1）找到 pending 了哪个信号。

（2）从内核态返回用户态的路上势必要弹掉该 task 以前的用户态 context，所以内核要把其保存下来。

（3）改变从普通的内核态返回用户态的路径，铺设一条能够返回到 Signal 处理函数的路径。

（4）Signal 处理函数还是用该 task 的用户栈，只是这个栈最新的 frame 是被内核杜撰出来的假环境。

4.2.1 路径切换

在中断返回、系统调用返回的路径中，最后一项工作是检测当前线程是否有待处理的信号，如果确有信号 pending，则中断返回用户态的路径，进入信号执行。

```
/*在 Signal pending 时被中断返回、系统调用返回调用，这里标志着信号 Handler 执行工作
  正式开始*/
asmlinkage void do_notify_resume(…)
{
    //进入该函数的参数 regs 指向当前栈顶，这里是内核栈保持用户态 context 地方
    if (thread_flags & _TIF_SIGPENDING) //检查当前 task 是否有信号 pending
        do_signal(&current->blocked, regs, syscall);
}

static int do_signal(sigset_t *oldset, struct pt_regs *regs, int syscall)
{
    …
    //通过检查保存的 cspr 的后四位来确定 regs 是否指对了
    if (!user_mode(regs))
        return 0;

    //休眠操作
    if (try_to_freeze())
```

```
        goto no_signal;

    single_step_clear(current);

//找出到底是 pending 了那种信号
    signr = get_signal_to_deliver(&info, &ka, regs, NULL);
    …

 no_signal:
   //这一部分参见系统调用重入
    return 0;
}
```

4.2.2　ARM Linux 下信号执行环境的搭建

Signal Handler 的执行不同于向用户态代码段的返回，那里有着保存完好的寄存器状态，有着完整栈轨迹，弹回去接着跑即可。但是 Signal Handler 虽然是一段编译好的代码，但是进入时处理器的寄存器状态却需要杜撰出来，而且内核也要确保其执行完能够调回来，所以要为其准备栈内容。

```
//Signal Handler 执行环境的准备
static void handle_signal(…)
{
    …
    /*若处在系统调用，需检查系统重入情况，参见下文*/
    if (syscall) {
        switch (regs->ARM_r0) {
        case -ERESTART_RESTARTBLOCK:
        case -ERESTARTNOHAND:
            …
        case -ERESTARTSYS:
            …
        case -ERESTARTNOINTR:
            …
    }

    …

    //信号处理函数的运营环境的搭建主要在这里进行
    if (ka->sa.sa_flags & SA_SIGINFO)
        ret = setup_rt_frame(usig, ka, info, oldset, regs);
    else
        ret = setup_frame(usig, ka, oldset, regs);
```

```
    …

}
```

为了搭建信号处理函数的运行环境，内核使用了如下结构（这是位于 task 用户态的 stack 上）：

```
struct sigframe {
    struct ucontext uc;
    unsigned long retcode[2];
};

struct ucontext {
    unsigned long    uc_flags;
    struct ucontext  *uc_link;
    stack_t        uc_stack;
    //存放 task 以前用户态的 context
    struct sigcontext uc_mcontext;
    sigset_t       uc_sigmask;
    …
    unsigned long    uc_regspace[128] __attribute__((__aligned__(8)));
};

//堆栈搭建
static int setup_frame(int usig, struct k_sigaction *ka, sigset_t *set,
struct pt_regs *regs)
{

    //首先找到 task 用户栈，然后从用户栈再分配出 sizeof(*frame)的空间
    struct sigframe __user *frame = get_sigframe(ka, regs, sizeof(*frame));
    …
    err |= setup_sigframe(frame, regs, set);
    if (err == 0)
        err = setup_return(regs, ka, frame->retcode, frame, usig);

    return err;
}

/*这里需要把内核栈里保存的 task 用户态 contex 保存在 struct sigcontext uc_mcontext 里*/
static int
setup_sigframe(struct sigframe __user *sf, struct pt_regs *regs, sigset_t
*set)
{
    ...
    //保存寄存器
```

```
    __put_user_error(regs->ARM_r0, &sf->uc.uc_mcontext.arm_r0, err);
    __put_user_error(regs->ARM_r1, &sf->uc.uc_mcontext.arm_r1, err);
    …
    __put_user_error(regs->ARM_r8, &sf->uc.uc_mcontext.arm_r8, err);
    …
    __put_user_error(regs->ARM_cpsr, &sf->uc.uc_mcontext.arm_cpsr, err);

    //保存线程相关状态
    __put_user_error(current->thread.trap_no, &sf->uc.uc_mcontext.trap_no,
    err);
    …
    __put_user_error(set->sig[0], &sf->uc.uc_mcontext.oldmask, err);
    err |= __copy_to_user(&sf->uc.uc_sigmask, set, sizeof(*set));
    …

    //保存 marvell  IWMMXT 协处理器 conetxt
#ifdef CONFIG_IWMMXT
    if (err == 0 && test_thread_flag(TIF_USING_IWMMXT))
        err |= preserve_iwmmxt_context(&aux->iwmmxt);
#endif

//这里保存 arm vfp 协处理器 conetxt

#ifdef CONFIG_VFP
    if (err == 0)
        err |= vfp_save_state(&sf->aux.vfp);
#endif
    __put_user_error(0, &aux->end_magic, err);

    return err;
}

//搭建跳转路径
static int setup_return(struct pt_regs *regs, struct k_sigaction *ka,
        unsigned long __user *rc, void __user *frame, int usig)
{
    //内核记录的信号处理函数的入口地址
    unsigned long handler = (unsigned long)ka->sa.sa_handler;
    unsigned long retcode;
    int thumb = 0;
    /*将 cspr 的条件状态位清零，ARM 指令是条件执行的，既然信号处理函数是不依赖以前的状
      态，自然需要将条件状态位清零*/
    unsigned long cpsr = regs->ARM_cpsr & ~PSR_f;

    if (ka->sa.sa_flags & SA_THIRTYTWO)
```

```
        cpsr = (cpsr & ~MODE_MASK) | USR_MODE;

    //这里略去 thumb 指令方式下的处理
    …

    if (ka->sa.sa_flags & SA_RESTORER) {
    //如果设置了信号处理函数返回内核的地址，就优先用
        retcode = (unsigned long)ka->sa.sa_restorer;
    } else {
        unsigned int idx = thumb << 1;

    /    /如果设置了 SA_SIGINFO，选择下面那个入口
        if (ka->sa.sa_flags & SA_SIGINFO)
            idx += 3;

        if (__put_user(sigreturn_codes[idx],   rc) ||
            __put_user(sigreturn_codes[idx+1], rc+1))
            return 1;

        if (cpsr & MODE32_BIT) {

            /*如果是普通 signal，idx 为零，选择第一个入口点;
            如果是 rt signal，idx 为 3 左移 2 位等于 12，正好是第二个入口点*/
            retcode = KERN_SIGRETURN_CODE + (idx << 2) + thumb;
        } else {
          …
            //Thumb 指令方式下的处理，这里不分析

        }
    }

    //作为信号处理函数的参数
    regs->ARM_r0 = usig;
   //信号处理函数新栈
    regs->ARM_sp = (unsigned long)frame;
   //返回入口地址，放到 lr，等到信号处理函数返回时使用
    regs->ARM_lr = retcode;
   //信号处理函数的入口地址
    regs->ARM_pc = handler;
   //条件状态位清零后的 cspr
    regs->ARM_cpsr = cpsr;

    return 0;
}
```

4.2.3　Signal 处理函数的返回

Signal 处理函数执行完毕之后需要返回到内核态，为了迎接其返回，内核做了如下安排。

内核在异常向量表+0x00000500 的地方放置了内核返回入口：

```
const unsigned long sigreturn_codes[7] = {
    MOV_R7_NR_SIGRETURN,    SWI_SYS_SIGRETURN,    SWI_THUMB_SIGRETURN,
    MOV_R7_NR_RT_SIGRETURN, SWI_SYS_RT_SIGRETURN, SWI_THUMB_RT_SIGRETURN,
};
```

这是一个若干指令组成的数组，该数组在 void __init early_trap_init(…)中和异常向量表一起被复制到最终位置。检查该数组发现，其中的指令其实是软中断调用，可见信号处理函数返回内核借助于软中断调用。再者内核中准备了两个系统调用 asmlinkage int sys_sigreturn(…)，asmlinkage int sys_rt_sigreturn(…)用来处理异常向量表+0x00000500 处的 2 个软中断调用。

```
asmlinkage int sys_sigreturn(struct pt_regs *regs)
{
    struct sigframe __user *frame;

    …
    current_thread_info()->restart_block.fn = do_no_restart_syscall;

    if (regs->ARM_sp & 7)
        goto badframe;

    // regs->ARM_sp 是信号处理函数的栈
    frame = (struct sigframe __user *)regs->ARM_sp;

    //由于其在用户态，需要做一次检查
    if (!access_ok(VERIFY_READ, frame, sizeof (*frame)))
        goto badframe;
    //恢复 task 进入信号处理函数之前的用户态
    if (restore_sigframe(regs, frame))
        goto badframe;
   …
    return regs->ARM_r0;

…
}

/*因为在 launch 异常处理函数之前就把 task 的用户态 context 做了保存，所以到了取出的
  时候*/
```

```
static int restore_sigframe(struct pt_regs *regs, struct sigframe __user
*sf)
{
    struct aux_sigframe __user *aux;
    sigset_t set;
    int err;

    /*除了 task 用户态的 context，当前 task 的信号 blocked 屏蔽位图也被保存，从用户态
      内存拷贝恢复*/
    err = __copy_from_user(&set, &sf->uc.uc_sigmask, sizeof(set));
    if (err == 0) {
        sigdelsetmask(&set, ~_BLOCKABLE);
        spin_lock_irq(&current->sighand->siglock);
        current->blocked = set;
        recalc_sigpending(); //重新计算当前是否可以清楚信号 pending 位
        spin_unlock_irq(&current->sighand->siglock);
    }
    //以下就是恢复 task 的用户态 context
    __get_user_error(regs->ARM_r0, &sf->uc.uc_mcontext.arm_r0, err);
    __get_user_error(regs->ARM_r1, &sf->uc.uc_mcontext.arm_r1, err);
    …
    __get_user_error(regs->ARM_sp, &sf->uc.uc_mcontext.arm_sp, err);
    __get_user_error(regs->ARM_lr, &sf->uc.uc_mcontext.arm_lr, err);
    __get_user_error(regs->ARM_pc, &sf->uc.uc_mcontext.arm_pc, err);
    __get_user_error(regs->ARM_cpsr, &sf->uc.uc_mcontext.arm_cpsr, err);

    err |= !valid_user_regs(regs);

    //这里保存的是特殊的诸如协处理器的 context
    aux = (struct aux_sigframe __user *) sf->uc.uc_regspace;

…
/*恢复 IWMMX 协处理器 context，对于 CA9 则需要处理 Neon 的 context，内核在切换、信号
  等方面对 Neon 的处理方式与 IWMMX 协处理器相同*/
#ifdef CONFIG_IWMMXT
    if (err == 0 && test_thread_flag(TIF_USING_IWMMXT))
        err |= restore_iwmmxt_context(&aux->iwmmxt);
#endif
…
    return err;
}
```

由于又恢复 task 用户态到内核栈，所以当内核从这里返回的时候顺着铺好的内核栈就自然而然的弹回到中断或者系统调用前的用户态地址。

4.2.4　系统调用重入

关于系统调用重入有以下两种情况。

（1）信号被屏蔽或者被忽略，这时不需要执行信号处理函数。

（2）需要执行信号处理函数。

先看第一种，若某个系统调用被打断，它返回时必会以如下错误代码返回：

如果以 block 方式重启 ：-ERESTARTNOHAND

如果以重新调用方式重启：-ERESTARTNOHAND，-ERESTARTSYS，-ERESTARTNOINTR

```
static int do_signal(sigset_t *oldset, struct pt_regs *regs, int syscall)
{
    //参见 signal 处理函数执行
   ...
no_signal:
    //如果是从系统调用过来的
    if (syscall) {
        if (regs->ARM_r0 == -ERESTART_RESTARTBLOCK) {
            /*Thumb 指令集的处理，明显指令长度较短，这里不做进一步展开*/
            if (thumb_mode(regs)) {
                regs->ARM_r7 = __NR_restart_syscall - __NR_SYSCALL_BASE;
                regs->ARM_pc -= 2;
            } else {

#if defined(CONFIG_AEABI) && !defined(CONFIG_OABI_COMPAT)
                // AEABI 编译器相关
                regs->ARM_r7 = __NR_restart_syscall;
                regs->ARM_pc -= 4;
#else
                u32 __user *usp;
                u32 swival = __NR_restart_syscall;

                //用态堆栈开出一个新的 frame
                regs->ARM_sp -= 12;
                //新的 frame
                usp = (u32 __user *)regs->ARM_sp;

                swival=swival-__NR_SYSCALL_BASE+__NR_OABI_SYSCALL_BASE;
                //保存 task 用户态的 pc 值
                put_user(regs->ARM_pc, &usp[0]);
                //将一个对系统调用 restart_syscall 的指令压栈
                /* swi __NR_restart_syscall */
                put_user(0xef000000 | swival, &usp[1]);
                //把如下指令压栈
```

```
                /* ldr  pc, [sp], #12 */
                put_user(0xe49df00c, &usp[2]);
                //改变了代码段，需要更新 icache
                flush_icache_range((unsigned long)usp,
                           (unsigned long)(usp + 3));
                /*这一行最重要，当回到用户态时，pc 的指针指向产生软中断的那条指令地址*/
                regs->ARM_pc = regs->ARM_sp + 4;
#endif
            }
        }
        if (regs->ARM_r0 == -ERESTARTNOHAND ||
            regs->ARM_r0 == -ERESTARTSYS ||
            regs->ARM_r0 == -ERESTARTNOINTR) {
            //参见下文的 setup_syscall_restart 分析
            setup_syscall_restart(regs);
        }
    }
    single_step_set(current);
    return 0;
}
//用户态 Context 的修改
static inline void setup_syscall_restart(struct pt_regs *regs)
{
    regs->ARM_r0 = regs->ARM_ORIG_r0;
/*把用户态 pc 指针后退一格，正好指向 swi 发生前的地址，将导致原系统调用被重新调用*/
    regs->ARM_pc -= thumb_mode(regs) ? 2 : 4;
}
```

重入口的定义如下：

```
SYSCALL_DEFINE0(restart_syscall)
{
    struct restart_block *restart = &current_thread_info()->restart_block;
    //重新开始没有完成的工作
    return restart->fn(restart);
}
```

第 5 章　进程与进程内存

内核最大的意义是将蛮荒的硬件世界改造成文明的进程、线程社会。

5.1　Linux 进程

5.1.1　Fork

笔者年轻时只关注内核，那时觉着 Linux 的先 Fork 再 Exec 机制太拖泥带水，一步创建全新进程的机制才够劲，应该抛弃老进程，直接为一个新进程构造一个全新的环境。在研究完 Android 操作系统之后才明白，Fork 机制才是构建大型操作系统的重要基石，若内核不支持资源继承的进程创建机制，是无法或者很难构建大型操作系统的。

（1）Fork 第一个重要工作是管理资源继承，这里面包括文件、信号、虚拟内存等方面的资源。其中虚拟内存的处理方式不同，将导致是产生新的进程还是当前进程内线程，以及新老进程之间的关系。

Fork 机制中关于诸如文件、信号等方面的资源处理对内核的后续影响比较直接，本书不进行展开，而关于虚拟内存的处理，本书以 Android Java 进程为例在下文分析。

（2）Fork 机制的另一个重要工作为新进程的第一个线程指定内核态的起始地址，从而为新进程建立的运行轨迹，由于新进程的第一个线程运行的时刻都是从第一次被调度处理器开始，该时刻处于内核态，然后顺着继承而来栈弹到 Fork 调用的下一条地址，当然那已经是新进程的虚拟内存空间了。

新进程的第一个线程生命的第一时刻是在调度器里开始的，其内核起始路径的指定是通过修改线程的内核切换 context 且准备一个新的没有老线程运行轨迹的内核栈来完成的，代码实现如下：

```
Int copy_thread(unsigned long clone_flags, unsigned long stack_start,
        unsigned long stk_sz, struct task_struct *p, struct pt_regs *regs)
{
    struct thread_info *thread = task_thread_info(p);
    //内核栈的栈底存放着对应线程用户态的 context
    struct pt_regs *childregs = task_pt_regs(p);

    *childregs = *regs;
   //子进程
    childregs->ARM_r0 = 0;
    /*用户态堆栈，对于新进程这就是老进程的用户栈，对于同一进程的新线程，线程库在这里存
```

```
    放新的线程入口*/
    childregs->ARM_sp = stack_start;
    //情况切换 context
    memset(&thread->cpu_context, 0, sizeof(struct cpu_context_save));
    //设置新进程或线程的内核栈
    thread->cpu_context.sp = (unsigned long)childregs;
    //设置新进程或线程的内核运行起点
    thread->cpu_context.pc = (unsigned long)ret_from_fork;
    …
    return 0;
}
```

往后新线程逐步退出内核态，转向用户态的执行 clone 调用的后一条地址。这是用户态起始地址的设置是由线程库实现的，关于线程库的实现请参考 Android 部分。

新进程继承了老进程虚拟地址，自然就继承其了用户栈，其转向用户态的第一条执行地址即为老进程做 Fork 调用的下一地址。

5.1.2 Exec 新进程创建

在 Android 系统里 Exec 机制虽然没有 Fork 机制那么重要，但却是系统中必不可少的组成部分。因为除了基于 Java 进程，系统中还有诸如 init、硬件设备用户层 deamon 之类的进程大量存在。

新进程创建分为以下两个主要方面。

（1）脱离老进程的 Context，Linux 内核没有提供直接创建一个新进程的机制，所有的新进程都是从上一个被 Fork 出来进程演进过来，所以新进程首先释放老进程虚拟地址空间、关闭老进程打开的文件、消灭老进程里除当前线程之外的线程等工作。

（2）打开新进程的二进制镜像、为新进程准备最初的运行栈、找出新进程的动态链接器并 MAP 之、为新进程建立虚拟地址空间、设置新进程用户态起跑点……

```
//Exec 构建新进程环境的主体
static int load_elf_binary(struct linux_binprm *bprm, struct pt_regs *regs)
{
    …
    elf_phdata = kmalloc(size, GFP_KERNEL);
    …
    //将二进制 ELF 镜像的头读进来
    retval = kernel_read(bprm->file, loc->elf_ex.e_phoff,
                (char *)elf_phdata, size);
    …

/*GCC 连接的时候会把使用的动态连接器文件路径放在 ELF 里，这里扫描 ELF 找到动态连接器*/
    for (i = 0; i < loc->elf_ex.e_phnum; i++) {
//检查这一段是否 PT_INTERP，如果是 PT_INTERP 表示这里藏着动态连接器
```

```
    if (elf_ppnt->p_type == PT_INTERP) {
        …
        // elf_interpreter 将存放路径字符串
        elf_interpreter = kmalloc(elf_ppnt->p_filesz,
                     GFP_KERNEL);
        …
        //将路径读进 elf_interpreter
        retval = kernel_read(bprm->file, elf_ppnt->p_offset,
                    elf_interpreter,
                    elf_ppnt->p_filesz);
        …
        /*打开动态连接器文件，对于 Android 就是/system/bin/linker，对与 ubuntu
          就是/lib/ld-x.x.x.so*/
        interpreter = open_exec(elf_interpreter);
        …
        /*读取动态连接器二进制镜像上的 BINPRM_BUF_SIZE 一段到 bprm->buf，动
          态链接器其实也是 ELF 文件，其实就是把其 ELF 头读进来*/
        retval = kernel_read(interpreter, 0, bprm->buf,
                    BINPRM_BUF_SIZE);
        …
        /* Get the exec headers */
        loc->interp_elf_ex = * ((struct elfhdr *)bprm->buf);
        break;
    }
    elf_ppnt++;
}

elf_ppnt = elf_phdata;
//可见是不是可执行栈是由编译时决定的
for (i = 0; i < loc->elf_ex.e_phnum; i++, elf_ppnt++)
    if (elf_ppnt->p_type == PT_GNU_STACK) {
        if (elf_ppnt->p_flags & PF_X)
            //可执行栈
            executable_stack = EXSTACK_ENABLE_X;
        else
            //不可执行栈，Android 的二进制程序属于这种情况
            executable_stack = EXSTACK_DISABLE_X;
        break;
    }

if (elf_interpreter) {
    //这里对动态链接器进行检查，略去
    …
}
//和过去的自己说再见，下文详细分析
```

```
    retval = flush_old_exec(bprm);
    …
    //还是和过去的自己说再见而且为新进程命名，在下文详细分析
    setup_new_exec(bprm);
    …
    //设置新进程的启动参数，攒出新进程的最初的用户栈
    retval = setup_arg_pages(bprm, randomize_stack_top(STACK_TOP),
                executable_stack);
    …

    current->mm->start_stack = bprm->p;
```

/*把二进制程序镜像 ELF 的 PT_LOAD 虚拟地址映射出来，在 Android 系统中以 system/bin/app_process 为例，该镜像的结构可以使用"prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin/arm-eabi-readelf -a out/target/product/generic/system/bin/app_process "dump 出来，其 Program Headers 结构节选如下：

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00008034 0x00008034 0x000e0 0x000e0 R   0x4
  INTERP         0x000114 0x00008114 0x00008114 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /system/bin/linker]
  LOAD           0x000000 0x00008000 0x00008000 0x01194 0x01194 R E 0x1000
  LOAD           0x002000 0x0000a000 0x0000a000 0x001b0 0x001c0 RW  0x1000
  DYNAMIC        0x002040 0x0000a040 0x0000a040 0x000e8 0x000e8 RW  0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0
  EXIDX          0x00114c 0x0000914c 0x0000914c 0x00048 0x00048 R   0x4
```

其中的 INTERP 类型的段就是存放动态链接器路径的，这个没有 MAP 出来，而是使用文件 read 操作读取出来了，而其中两个 LOAD 类型的段，其起始地址分别位于 0x00008000 和 0x0000a000。下面内核需要将这两个段 MAP 出来*/

```
    for(i = 0, elf_ppnt = elf_phdata;
        i < loc->elf_ex.e_phnum; i++, elf_ppnt++) {
        int elf_prot = 0, elf_flags;
        unsigned long k, vaddr;
        //找 PT_LOAD 类型的段，别的不管
        if (elf_ppnt->p_type != PT_LOAD)
            continue;

        //每一次 MAP 之前都要看是不是要做 brk
        if (unlikely (elf_brk > elf_bss)) {
            unsigned long nbyte;
            retval = set_brk(elf_bss + load_bias,
                     elf_brk + load_bias);
```

```
        ...

    }
    /*检索该段的属性，并记录在 elf_prot 中，这个要在 MAP 时使用*/
    if (elf_ppnt->p_flags & PF_R)
        elf_prot |= PROT_READ;
    if (elf_ppnt->p_flags & PF_W)
        elf_prot |= PROT_WRITE;
    if (elf_ppnt->p_flags & PF_X)
        elf_prot |= PROT_EXEC;

    elf_flags = MAP_PRIVATE | MAP_DENYWRITE | MAP_EXECUTABLE;

    vaddr = elf_ppnt->p_vaddr;
    if (loc->elf_ex.e_type == ET_EXEC || load_addr_set) {
        //0x00008000 和 0x0000a000 两个段都是到这
        elf_flags |= MAP_FIXED;
    } else if (loc->elf_ex.e_type == ET_DYN) {
        ...
    }
    /*MAP 动作，这里仅仅跟进程挂上 struct vm_area_struct 描述结构，没有页表页
      目录操作，要等到程序运行产生也异常才会发生*/
    error = elf_map(bprm->file, load_bias + vaddr, elf_ppnt,
            elf_prot, elf_flags, 0);
    ...
    }

//结果检查
...
}
...
/*这里要做工作是把动态链接器的需要 MAP 出来的段 MAP 出来，后面详细分析*/
if (elf_interpreter) {
    unsigned long uninitialized_var(interp_map_addr);

    elf_entry = load_elf_interp(&loc->interp_elf_ex,
                    interpreter,
                    &interp_map_addr,
                    load_bias);
/*对于 Android 系统的 system/bin/linker，elf_entry 返回值为 0*/
    if (!IS_ERR((void *)elf_entry)) {

        /* elf_entry 是整个程序用户态的入口地址，然而它却不在程序二进制的虚拟空
           间上，而是在动态连接器的入口点上，对于 system/bin/app_process，其
           ELF 头的 loc->interp_elf_ex.e_entry 里面记录了这个入口点的地址：
```

```
                0x b0001000*/
            interp_load_addr = elf_entry;
            elf_entry += loc->interp_elf_ex.e_entry;
        }
        …
    } else {
        …
    }

    …
    //新进程 struct cred 初始化
    install_exec_creds(bprm);
    //代码数据段起始结束地址在这里设置
    /* N.B. passed_fileno might not be initialized? */
    current->mm->end_code = end_code;
    current->mm->start_code = start_code;
    current->mm->start_data = start_data;
    current->mm->end_data = end_data;
    current->mm->start_stack = bprm->p;

…
/*为新进程准备用户态的返回点，新进程一直使用 Exec 之前的内核栈在忙活，然后它顺着个内核
栈往回跑，知道要返回用户态时，其 pc 被指向了新的地址，用户栈也被换掉了，所以当其 Exec
之后第一次返回用户态时就走向了新生。这里要做的是：
(1) 设置其用户态 pc 地址—动态链接器的入口地址。
(2) 设置用户栈的地址，动态链接器要根据这个最初的用户栈地址找到启动参数，找到二进制程序
的入口点
*/

    start_thread(regs, elf_entry, bprm->p);
    …
}
```

动态链接器的加载类似于二进制镜像的加载，也是要 MAP PT_LOAD 类型的段，以 Android 的 system/bin/linker 为例分析。首先使用命令：prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin/arm-eabi-readelf -a out/target/product/generic/system/bin/linker dump system/bin/linker 的结构，其 Program Headers 节选如下：

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x0000d4 0x00000000 0xb0000000 0x00000 0x00000 R   0x1000
  LOAD           0x001000 0xb0001000 0xb0001000 0x0adcc 0x0adcc R E 0x1000
  LOAD           0x00c000 0xb000c000 0xb000c000 0x006fc 0x0b4d0 RW  0x1000
  GNU_STACK       0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0
  EXIDX          0x00ba84 0xb000ba84 0xb000ba84 0x00348 0x00348 R   0x4
```

可见要 MAP 出 0xb0001000 和 0xb000c000 这两个段。其中 0xB0001000 是 system/bin/linker 的基地址，也是其入口处，参见其 makefile 文件 bionic/linker/Android.mk：

```
LINKER_TEXT_BASE := 0xB0001000
```

关于 system/bin/linker 的实现参见 Android 章节，本节着重于内核机制的分析。动态链接器的加载在函数 static unsigned long load_elf_interp(…)里进行，与程序二进制镜像加载非常相似，可参见程序二进制镜像加载，不再赘述。

5.2　CPU 与 MMU

配备 MMU 的处理器是复杂系统运行的关键，虚拟内存的重要性在计算机体系结构里的作用无论怎么强调都不算过分。虚拟内存机制的贡献在于解决如下两大问题。

1. 内存碎片的困扰

一个复杂应用跑起来经常做的动作是什么？是内存的申请和释放。一旦一块内存被占用，意味着该内存块之前之后的内存被分割开。随着应用的复杂度增加，内存被严重地碎片化，导致系统后来很难找的合适长度内存以满足应用的需要。

2. 大量的内存浪费

在没有虚拟内存的支持的系统下，所有的内存操作都是真实的物理内存，而且每次申请都需要最大限度满足需要，申请一个 16K 的栈，无论栈空栈满都要提前满足，如果只分配 8K，等到栈到 8K 的时候，后面的物理内存可能已经无法使用了（碎片化或者被占用）。加载一个二进制镜像时需要全部加载进内存，不然当跑到代码段页面异常时，系统是无法请页的。

5.2.1　ARM Linux 页表页目录结构

32 位处理器的 MMU 的页表层次通常为两级，ARM V7 也不例外，有些叫法为页目录、页表，为了简单起见称之为 L1 页表 L2 页表。为了对于两级页表的处理器在其页表定义文件 arch/arm/include/asm/pgtable.h 中包含文件#include <asm-generic/4level-fixup.h>，该文件屏蔽了三级页表的操作。

基本页表操作定义如下：

```
/*索引虚拟地址 addr 对于的 L1 页表项*/
#define pgd_offset(mm, addr)    ((mm)->pgd + pgd_index(addr))

/*在 ARM 32 位架构下，该宏直接定义为 L1 的页表项*/
#define pud_alloc(mm, pgd, address) (pgd)
```

```
/*PMD 操作对 ARM 32 位无效，在 ARM 32 位体系下#define pud_none(pud)   0，#define
pmd_offset(dir, addr)   ((pmd_t *)(dir))，所以该宏直接返回 pud*/
#define pmd_alloc(mm, pud, address) \
    ((unlikely(pgd_none(* (pud))) && __pmd_alloc(mm, pud, address))? \
        NULL: pmd_offset(pud, address))
```

在 ARM Linux 系统中有两种页表结构，一种是 3G 以上的一级页表，一种是上一节提到的二级页表。前者是内核态的寻址方式，是静态的。后者是应用虚拟地址使用寻址方式，是动态的。

Linux 操作系统从 3G 开始，把虚拟地址沿着物理内存的开始，一直映射到物理内存的边界（这就是 Linux 系统上物理内存超过 1G 遇到诸多问题的原因，不在于处理器转换不了，而是 OS 设计的问题）。这部分页表只起到虚拟地址到物理地址转换的作用，通常不产生异常。为了节省转换时间，提高 TLB 利用率，Linux 内核借助 ARM MMU 的 1M 段转换项，直接在 L1 级页表中转换物理地址。

3G 以下的地址采用上节提到的二级页表模型，不过为了使得 ARM MMU 能够在 Linux 的虚拟内存框架下工作，做了如下设计：

在 Linux 虚拟内存模型下，需要记录页面的 dirty、可读、可写等属性。但是 ARM L2 的页表项的结构并不能跟 Linux 的要求一一对应，而且 L2 页表项又被硬件全部占满。所以对于每个 L2 页表，Linux 准备了两份，一份给 Linux 用，一份给硬件 MMU 用。其组织结构是在一个 4K 大小的页面上。

256 项页表项的 Linux 页表 0|256 项页表项的 Linux 页表 1|256 项页表项的 MMU 页表 0|256 项页表项的 MMU 页表 1

这样每次分配 L2 的页表必须分配两个，代码分析如下：

```
/*L2 页表的分配与挂接，mm 为当前 task 的 struct mm_struct，vma 为异常虚拟地址对应的
  虚拟内存描述结构，pmd 对于 ARM 32 位就是 L1 的页表项，address 为发生异常的虚拟地址 */
int __pte_alloc(struct mm_struct *mm, struct vm_area_struct *vma,
        pmd_t *pmd, unsigned long address)
{   //分配 L2 页表
    pgtable_t new = pte_alloc_one(mm, address);
    …
    /*分配完了 L2 页表，下面要把 L2 页表挂载在 L1 对应的页表项中，对当前进程的页表进行操
      作，这时别的处理器上的或者当前处理器上发生抢占的当前进程里其他的线程可以也会发生
      页异常，而且地址可能跟 address 在同一个 4K 里（对于 ARM 是同一个 8K）将会产生冲突，
      所以要锁住 page_table_lock */
    spin_lock(&mm->page_table_lock);
    …
    /*检查一下对应的 L1 是否已经被填上，从发生异常到锁住 page_table_lock 这段时间可能
      有个哥们已经把活干完了，如果是这样那就省事了，把分配的页表释放掉即可*/
    if (likely(pmd_none(*pmd))) {   /* Has another populated it ? */
        mm->nr_ptes++;
    /*把 L2 挂到 L1 上*/
```

```
        pmd_populate(mm, pmd, new);
        new = NULL;
    } else if (unlikely(pmd_trans_splitting(*pmd)))//ARM32 位不考虑这种情况
        wait_split_huge_page = 1;
    /*开锁*/
    spin_unlock(&mm->page_table_lock);
    /*对应从发生异常到锁住 page_table_lock 这段时间另外的线程已经修复了页表，把申请
      的页表释放掉*/
    if (new)
        pte_free(mm, new);
    …
    return 0;
}
/*把 L2 挂到 L1*/
static inline void __pmd_populate(pmd_t *pmdp, phys_addr_t pte,
    unsigned long prot)
{
    /* pte 是分配 4K 大小的内存的物理地址，前 2K 留给 Linux 使用。PTE_HWTABLE_OFF
       =512x4，prot 是属性信息*/
    unsigned long pmdval = (pte + PTE_HWTABLE_OFF) | prot;
    /* pmdval 执行了 MMU 硬件使用的页表，填到 L1 的页表项*/
    pmdp[0] = __pmd(pmdval);
    /* L2 的页表项数目是 256，往下再拉 256 项，指向第二张页表*/
    pmdp[1] = __pmd(pmdval + 256 * sizeof(pte_t));
    flush_pmd_entry(pmdp);
}
```

在内核需要设置 L2 的页表项时，调用如下函数：

```
extern void cpu_set_pte_ext(pte_t *ptep, pte_t pte, unsigned int ext);
#define cpu_set_pte_ext(ptep,pte,ext) processor.set_pte_ext(ptep,pte,ext)
```

可见最终，L2 页表项的设置是架构相关支持代码，位于 proc-xx.S 中，v7 架构相关代码如下：

```
/*R0 里存放内核页表项地址，R1 里存放内核页表项值，R2 指出硬件版本页表项值的第 11 位属
     性值*/
ENTRY(cpu_v7_set_pte_ext)
#ifdef CONFIG_MMU
    /*直接写入内核使用的页表项信息，这仅能够被 Linux 内核访问到，ARM MMU 不会使用*/
    str r1, [r0]            @ linux version
    /*根据内核使用页表项信息，配置硬件使用的页表项与 ARM 处理的 MMU 有着复杂灵活的配置
      和使用，Linux 不可能把其所有的功能都发挥出来（事实上任何一种操作系统的 ARM 处理
      器实现也只需且只是利用其部分功能）。关于 ARM MMU 协处理器的研究本书不做展开，感
      兴趣的读者请参见 infocenter.arm.com */
    bic r3, r1, #0x000003f0
```

```
    bic r3, r3, #PTE_TYPE_MASK
    orr r3, r3, r2
    orr r3, r3, #PTE_EXT_AP0 | 2

    tst r1, #1 << 4
    orrne   r3, r3, #PTE_EXT_TEX(1)
    //Dirty 属性的设置
    eor r1, r1, #L_PTE_DIRTY
    tst r1, #L_PTE_RDONLY | L_PTE_DIRTY
    orrne   r3, r3, #PTE_EXT_APX
    //特权级属性的设置
    tst r1, #L_PTE_USER
    orrne   r3, r3, #PTE_EXT_AP1
/*内核配置决定了是否使用 ARM MMU 的 domain 机制*/
#ifdef CONFIG_CPU_USE_DOMAINS
    tstne   r3, #PTE_EXT_APX
    bicne   r3, r3, #PTE_EXT_APX | PTE_EXT_AP0
#endif
   //执行属性的设置
    tst r1, #L_PTE_XN
    orrne   r3, r3, #PTE_EXT_XN

    tst r1, #L_PTE_YOUNG
    tstne   r1, #L_PTE_PRESENT
    moveq   r3, #0
/*将硬件使用的页表项值写入 L2 的页表项，这里是可以被 ARM 的 MMU 从 L1 索引过来*/
 ARM(   str r3, [r0, #2048]! )
 THUMB( add r0, r0, #2048 )
 THUMB( str r3, [r0] )
    mcr p15, 0, r0, c7, c10, 1      @ flush_pte
#endif
    mov pc, lr
ENDPROC(cpu_v7_set_pte_ext)
```

5.2.2 页表页目录的建立

页异常的发生有两种可能：一种是应用虚拟地址对应物理页的缺页、访问异常，这属于缺页与请页机制的范畴；一种是页表页目录转换过程的异常，这种异常导致是由页表页目录建立的。本节讨论后面一种。

```
/*该函数用于 Linux generic 的异常处理，支持多种架构和多种应用环境，里面不被 Android +
  Arm CA9 使用的代码都以斜体处理。该函数的主要作用是 L1 L2 页表项的索引与建立，并把属
  于最后一级页表项的异常引向缺页与请页机制*/
int handle_mm_fault(struct mm_struct *mm, struct vm_area_struct *vma,
```

```
        unsigned long address, unsigned int flags)
{
 /* 因为这里是抽象的 Linux 虚拟内存异常处理，要支持各种处理器架构，在某些 64 位的处理器
    架构下表达到 3、4 级，这就是如下 4 个页表项指针定义的原因，但是在 ARM 32 位下，前三
    个都是退化 L1 页表项*/
    pgd_t *pgd;
    pud_t *pud;
    pmd_t *pmd;
    pte_t *pte;
   …
   /*HUGETLB_PAGE 通过加大页面的颗粒度，从而提高 TLB 的命中率，在当前 CA9 架构下的
     Android 系统不考虑这个应用 */
    if (unlikely(is_vm_hugetlb_page(vma)))
        return hugetlb_fault(mm, vma, address, flags);

    pgd = pgd_offset(mm, address);
    pud = pud_alloc(mm, pgd, address);
    if (!pud)
        return VM_FAULT_OOM;
    pmd = pmd_alloc(mm, pud, address);
    if (!pmd)
        return VM_FAULT_OOM;
   /* Transparent Hugepages 依赖 X86 系统，与 Arm Android 无关*/
    if (pmd_none(*pmd) && transparent_hugepage_enabled(vma)) {
        if (!vma->vm_ops)
            return do_huge_pmd_anonymous_page(mm, vma, address,
                                  pmd, flags);
    } else {
        pmd_t orig_pmd = *pmd;
        barrier();
        /*仍然是 Transparent Hugepages 相关，不考虑*/
        if (pmd_trans_huge(orig_pmd)) {
            if (flags & FAULT_FLAG_WRITE &&
                !pmd_write(orig_pmd) &&
                !pmd_trans_splitting(orig_pmd))
                return do_huge_pmd_wp_page(mm, vma, address,
                                  pmd, orig_pmd);
            return 0;
        }
    }

    /*
    如果 L1 的页表项不存在，则要创建 L2 页表，并将其挂到 L1 的页表项，对于 ARM 32 位 L2
    页表是成对分配的
     */
```

```
    if (unlikely(pmd_none(*pmd)) && __pte_alloc(mm, vma, pmd, address))
        return VM_FAULT_OOM;
    /* if an huge pmd materialized from under us just retry later */
    if (unlikely(pmd_trans_huge(*pmd)))
        return 0;
    /*
     找到该虚拟地址对应的 L2 的页表项，这里取出的是 L2 的 Linux 页表项
     */
    pte = pte_offset_map(pmd, address);
    return handle_pte_fault(mm, vma, address, pte, pmd, flags);
}
```

L2 页表的索引如下：

```
#define pte_offset_map(pmd,addr)    (__pte_map(pmd) + pte_index(addr))
/*尽管 L1 页表项指向 4K 的后半段，但是这里是取得 4K 页对齐，再取映射的虚拟地址，所以又回
  到页起始边界上*/
#define __pte_map(pmd)       (pte_t *)kmap_atomic(pmd_page(* (pmd)))

/*虚拟地址除以 4K 得出是第几页，再与上 512-1，得出自己的在这连续的两个 L2 页表的索引
  位置*/
#define pte_index(addr)      (((addr) >> PAGE_SHIFT) & (PTRS_PER_PTE-1))
```

内核为 Linux 使用的 L2 页表定义了如下属性：

```
#define L_PTE_PRESENT        (_AT(pteval_t, 1) << 0)
#define L_PTE_YOUNG          (_AT(pteval_t, 1) << 1)
…
#define L_PTE_SHARED (_AT(pteval_t, 1) << 10)/ * shared(v6),coherent(xsc3)
*/
```

MMU 硬件使用的 L2 页表项完全满足 ARM 硬件规范，定义如下：

```
#define L_PTE_MT_UNCACHED   (_AT(pteval_t, 0x00) << 2)  /* 0000 */
#define L_PTE_MT_BUFFERABLE (_AT(pteval_t, 0x01) << 2)  /* 0001 */
#define L_PTE_MT_WRITETHROUGH   (_AT(pteval_t, 0x02) << 2)  /* 0010 */
…
#define L_PTE_MT_DEV_CACHED (_AT(pteval_t, 0x0b) << 2)  /* 1011 */
#define L_PTE_MT_MASK       (_AT(pteval_t, 0x0f) << 2)
```

5.3 进程虚拟内存

5.3.1 Android 进程虚拟内存的继承

进程虚拟内存空间包括内核态空间和用户态空间。其中内核态地址空间是所有进程的

共享页表页目录。这里研究的是用户态虚拟内存空间的创建，而且专指在 Android 系统中的 Java 进程。

Dalvik 首先通过 Fork 机制复制 DVM 虚拟机，然后基于 Android Application framework 加载相关的.so、activity、service……，于是 Android Java 进程就诞生了。

```
/*Fork 机制关于进程虚拟内存空间处理函数，参数 unsigned long clone_flags，指明了是
创建新进程还是新线程，参数 struct task_struct * tsk 是新的进程或线程*/
static int copy_mm(unsigned long clone_flags, struct task_struct * tsk)
{
    …
    /* Android 进程 Fork 的 clone_flags 参数是 0x11，而 DVM 线程 Fork 的 clone_flags
      参数是 0x 450f00*/

    if (clone_flags & CLONE_VM) {
    /* CLONE_VM 定义为#define CLONE_VM 0x00000100。而 DVM Fork clone_flags
       是 0x 450f00，所以 DVM 的线程创建走到这里，可见对于线程创建，struct mm_struct
       和页表页目录压根就是共享的，当然要引用计数的增加*/
        atomic_inc(&oldmm->mm_users);
        mm = oldmm;
        goto good_mm;
    }
    /*DVM 进程创建 clone_flags 是 0x11，走到这里，进行著名的 dup_mm(…)操作。创建一
      个新的进程，为其准备一个新的 struct mm_struct 结构，见下文*/
    retval = -ENOMEM;
    mm = dup_mm(tsk);
    …
    //新进程与其 struct mm_struct 结构匹配起来
    tsk->mm = mm;
    tsk->active_mm = mm;
    return 0;
    …
}

/* 虚拟内存空间的框架函数为新的进程创建其虚拟地址空间，这包括虚拟机地址转换机构页表页
   目录及虚拟地址段的描述结构 struct vm_area_struct 的继承与创建。新老进程最大的不
   同在于其数据段的完全隔离与新生，所以这里最值得关注的是 COW 的处理*/
struct mm_struct *dup_mm(struct task_struct *tsk)
{
    …
    /*给新进程分配 struct mm_struct 结构*/
    mm = allocate_mm();
    …
    /*新进程的 struct mm_struct 和 struct task_struct 建立关系*/
    if (!mm_init(mm, tsk))
        goto fail_nomem;
```

```
...
    /*struct vm_area_struct 链表和页表页目录的复制就在这里完成*/
    err = dup_mmap(mm, oldmm);
    ...
}
/*虚拟地址段 struct vm_area_struct 的复制*/
static int dup_mmap(struct mm_struct *mm, struct mm_struct *oldmm)
{
    ...
    /*如果要对 struct vm_area_struct 链表和页表页目录操作，必先取得 struct mm_struct
      的 struct rw_semaphore mmap_sem; */
    down_write(&oldmm->mmap_sem);
    /*前面一步保证了其他 CPU 上不会再有改动 struct vm_area_struct 链表和页表页目录
      操作，下一步要保证 L2 cache 里的内容跟内存的内容一致：flush cache*/
    flush_cache_dup_mm(oldmm);
    //对新进程的 struct mm_struct 做相关的初始化操作
   ...
    //实际工作在这里进行
    for (mpnt = oldmm->mmap; mpnt; mpnt = mpnt->vm_next) {
        struct file *file;
    /*每个 struct vm_area_struct 结构代表进程的一段虚拟地址，逐段取出当前进程的
      struct vm_area_struct 结构，该结构详细描述了某段虚拟地址起始地址、属性等信息*/

        if (mpnt->vm_flags & VM_DONTCOPY) {
            /*如果当前这段虚拟内存是不能复制的，那就直接跳到下一段虚拟内存*/
            long pages = vma_pages(mpnt);
            mm->total_vm -= pages;
            vm_stat_account(mm, mpnt->vm_flags, mpnt->vm_file,-pages);
            continue;
        }
        charge = 0;
        if (mpnt->vm_flags & VM_ACCOUNT) {
            /*该段地址属于 VM ACCOUNT 信息关注的范围。基本上这些分段地址都属于关注
              范围*/
            unsigned int len = (mpnt->vm_end - mpnt->vm_start) >> PAGE_SHIFT;
            if (security_vm_enough_memory(len))
                goto fail_nomem;
            charge = len;
        }
        /*为新进程分配 struct vm_area_struct 结构*/
        tmp = kmem_cache_alloc(vm_area_cachep, GFP_KERNEL);
        ...
        tmp->vm_flags &= ~VM_LOCKED;
        tmp->vm_next = tmp->vm_prev = NULL;
```

```
        //检查是否有 backed file
        file = tmp->vm_file;
        if (file) {
            //该段地址是 filebacked 类型
            struct inode *inode = file->f_path.dentry->d_inode;
            /*struct address_space *mapping 记录了该 file 的 page cache*/
            struct address_space *mapping = file->f_mapping;

            get_file(file);
            //检查该段地址的属性
            if (tmp->vm_flags & VM_DENYWRITE)
                atomic_dec(&inode->i_writecount);
            mutex_lock(&mapping->i_mmap_mutex);
            if (tmp->vm_flags & VM_SHARED)
                mapping->i_mmap_writable++;
            flush_dcache_mmap_lock(mapping);
            /*将该段 struct vm_area_struct 和 page cache 树链接起来*/
            vma_prio_tree_add(tmp, mpnt);
            flush_dcache_mmap_unlock(mapping);
            mutex_unlock(&mapping->i_mmap_mutex);
        }
       …
        /*将“struct vm_area_struct”结构挂到新进程的链表中*/
        …
        __vma_link_rb(mm, tmp, rb_link, rb_parent);
        rb_link = &tmp->vm_rb.rb_right;
        rb_parent = &tmp->vm_rb;

        mm->map_count++;
        /*这里进行该段内存对应的页表页目录的复制*/
        retval = copy_page_range(mm, oldmm, mpnt);
       …
    }
       …
}


//页表页目录的复制函数
int copy_page_range(struct mm_struct *dst_mm, struct mm_struct *src_mm,
        struct vm_area_struct *vma)
{
    …
    //新老 PGD
    dst_pgd = pgd_offset(dst_mm, addr);
```

```
    src_pgd = pgd_offset(src_mm, addr);
    //按地址复制页表页目录
    do {
        next = pgd_addr_end(addr, end);
        if (pgd_none_or_clear_bad(src_pgd))
            continue;
        /*复制页表页目录的层级结构，并不是简单的复制，会根据该段地址的属性进行相关操作*/
        if (unlikely(copy_pud_range(dst_mm, src_mm, dst_pgd, src_pgd,
                        vma, addr, next))) {
            ret = -ENOMEM;
            break;
        }
    } while (dst_pgd++, src_pgd++, addr = next, addr != end);

    …
}

/*复制页表项 PTE，即最小单位的虚拟地址复制，这里重点关心 COW 内存的处理 */
static inline unsigned long  copy_one_pte(struct mm_struct *dst_mm, struct
mm_struct *src_mm,
        pte_t *dst_pte, pte_t *src_pte, struct vm_area_struct *vma,
        unsigned long addr, int *rss)
{
    //当前进程和虚拟地址段的属性
    unsigned long vm_flags = vma->vm_flags;
    pte_t pte = *src_pte;
    struct page *page;

    …
    /*
    如果是 COW 内存段，pte 复制前需加上写保护，等于在 Fork 这一时刻将数据空间在新老进程
    里都锁住了一样
     */
    if (is_cow_mapping(vm_flags)) {
        //锁父进程数据
        ptep_set_wrprotect(src_mm, addr, src_pte);
        //锁子进程数据
        pte = pte_wrprotect(pte);
    }
   …

out_set_pte:
    //设置到新进程的页表项
    set_pte_at(dst_mm, addr, dst_pte, pte);
    return 0;
}
```

5.3.2　进程虚拟地址空间的获得

进程主要通过 map 机制获得其虚拟内存，而 Linux 管理虚拟内存地址空间的基本手段是通过 struct vm_area_struct 进行分段管理。这里有些面向对象的概念，相同属性的连续虚拟地址段由对应一个 struct vm_area_struct 结构，里面记录了该段地址的属性、起始地址、操作函数等内容。当异常发生时就索引到对应的 struct vm_area_struct 结构并使用其中的操作函数进行处理。

这里值得关注的是，在 Linux 体系下虚拟地址分为匿名和 file backed 两类。前者的内容只能存在于物理内存中，后者的内容可以在存储设备的文件上。一个基于 Linux 内核良好架构的操作系统的设计需要充分利用 file backed 内存机制。因为这可以灵活地调整物理内存的使用情况。典型的例子是 Android 系统的 Dalvik 代码段被 file map 到 Dalvik 进程中，这样不同的 Dalvik 进程可以共享一份位于 page cache 的 Dalvik 代码，而且对于被较少应用使用的 Android 类库占用物理内存的机会大大减少。

```
//虚拟地址空间的获取
unsigned long mmap_region(struct file *file, unsigned long addr,
              unsigned long len, unsigned long flags,
              vm_flags_t vm_flags, unsigned long pgoff)
{
    ...
    /*
    在企图寻找一个存在的 struct vm_area_struct 结构并把当前虚拟地址段合并进去的尝试
    失败以后创建新的 struct vm_area_struct 结构，开始一段新的地址
     */
    vma = kmem_cache_zalloc(vm_area_cachep, GFP_KERNEL);
    ...
    //新的 struct vm_area_struct 结构
    vma->vm_mm = mm;
    //起始地址
    vma->vm_start = addr;
    //结束地址
    vma->vm_end = addr + len;
    //记录诸如堆栈生长方式、COW 之类的属性
    vma->vm_flags = vm_flags;
    //计算页表项时使用
    vma->vm_page_prot = vm_get_page_prot(vm_flags);
    //文件 map 地址
    vma->vm_pgoff = pgoff;
    INIT_LIST_HEAD(&vma->anon_vma_chain);
    //file backed 文件的操作
    if (file) {
        error =-EINVAL;
```

```
        /*map 的文件只能发生改动，不能发生增减，若内存有此要求则无法做 file map 操作*/
        if (vm_flags & (VM_GROWSDOWN|VM_GROWSUP))
            goto free_vma;
        if (vm_flags & VM_DENYWRITE) {
            /*若内存是可写的但文件不可写，也不能发生映射*/
            error = deny_write_access(file);
            if (error)
                goto free_vma;
            correct_wcount = 1;
        }
        /*检查完毕，进行文件 map*/
        vma->vm_file = file;
        get_file(file);
        /*如何 map 文件是由具体的文件系统决定。对于 ext4 文件系统，这里执行其 static
          int ext4_file_mmap(…)函数，把该 struct vm_area_struct 的成员变量
          struct vm_operations_struct 设置为 struct vm_operations_struct
          ext4_file_vm_ops。对于 struct vm_operations_struct 结构最重要是提
          供异常处理函数和写函数，对于 ext4 分别是 int filemap_fault(…)和 int
          ext4_page_mkwrite(…) */
        error = file->f_op->mmap(file, vma);
        if (error)
            goto unmap_and_free_vma;
        …
    } else if (vm_flags & VM_SHARED) {
        //共享内存的处理
        error = shmem_zero_setup(vma);
        if (error)
            goto free_vma;
    }

    …
    /*将新建的 struct vm_area_struct 结构插入到进程的 struct vm_area_struct 树中*/
    vma_link(mm, vma, prev, rb_link, rb_parent);
    file = vma->vm_file;
…
}
```

第 6 章　缺页请页与内存 Shrink

缺页请页与内存 Shrink 机制属于进程虚拟地址的一部分，但是这部分内容非常重要，单独提取一章分析。因为从系统角度来看，内核的运行有两个动力之源，一是代码执行枢纽——调度器，根据系统发生的各种事件使得各种进线程被调度或睡眠；另外一个是数据处理枢纽——缺页请页与内存 Shrink 机制。

应用程序运行时获取的内存都是虚拟内存，不论用 malloc 分配出多大的内存，本质上不过就是 struct vm_area_struct 虚拟内存的长度记录的数值增长而已，没有物理内存的分配，也没有页表页目录的填充。应用程序获得物理内存的方式是其访问到了 malloc 分配内存地址。在该段内存没有映射到物理内存的情况下发生了页异常，从而才会导致物理内存的分配。

事实上，系统运行时没有这么绝对，大部分的 malloc 的实现都是先进行大块虚拟地址的 map 而导致 struct vm_area_struct 的改变，再分割成进行小规模的 chunck 以满足 malloc 的需求。所以在一个页的范围内，若别的 chunk 先发生了页异常，则该页内的其他 chunk 即使没有访问到，也有了对应的物理内存。这里可以参见本书关于 Android 系统相关章节的分析。

内存 Shrink 是缺页请页的反方向动作，其关键作用是物理内存紧张时决定释放多少内存、释放那些内存、如何释放内存、释放掉有用内存在请页时如何找回来。

6.1　缺页与请页

应用运行时产生大量的缺页异常触发系统中内存与存储设备之间大量数据交换。由此形成了存储介质到处理器的双向数据流，那么这个双向数据流由如下节点组成：

```
CPU <-->VM <-->VFS<--> FS <-->Stroage
```

1．读方向

飞奔的处理器对数据和指令的需求产生频繁的页异常，这些页异常被传送到 VM，VM 根据异常的地址和对应的文件描述符通知 VFS，把需要的数据读到 Page cache 里。VFS 紧接着通过具体的文件系统定位需要读取的介质位置，然后把读取请求发送到具体文件系统，最后由具体文件系统启动驱动把数据读进内存。

2．写方向

处理器通过 VM 或者直接把数据写入 Page cache，然后 CPU 马不停蹄地去忙别的事情。

VM 跟 VFS 配合工作将这些数据提交具体文件系统，然后启动设备驱动写入外部存储。

两个方向的数据流都存在越过 VM，而直接操作 VFS 的文件读写路径，但这不是系统工作的主要方式，这条数据流的主要动力是缺页与请页机制。

6.1.1 File backed 虚拟内存段操作函数

虚拟内存段管理结构 struct vm_area_struct 是管理应用虚拟内存的中枢。当某段虚拟内存由于没有映射到对应的物理内存而发生异常时，要通过该结构的 int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);成员函数来处理。下面仅分析 Ext4 的实现：

```
/*第一个参数记录了异常发生时对应的 struct vm_area_struct 节点，第二个参数记了异常发
  生的虚拟地址。该函数的任务是找到虚拟地址对应的物理页*/
int filemap_fault(struct vm_area_struct *vma, struct vm_fault *vmf)
{
    int error;
    ...
    /*首先在该文件的 page cache 里寻找*/
    page = find_get_page(mapping, offset);
    if (likely(page)) {
        /*
        在该文件的 page cache 发现对应的物理页，进一步启动 async 预读。因为预期到系
        统很可能需要访问接下来的文件内容。比如 Dalvik 在执行代码时，很可能顺着该地址
        跑若干页
         */
        do_async_mmap_readahead(vma, ra, file, page, offset);
    } else {
        /* 没有在该文件的 page cache 发现对应的物理页，进一步启动 sync 预读，这里可
           能发生睡眠*/
        do_sync_mmap_readahead(vma, ra, file, offset);
        count_vm_event(PGMAJFAULT);
        mem_cgroup_count_vm_event(vma->vm_mm, PGMAJFAULT);
        ret = VM_FAULT_MAJOR;
retry_find:
      /*sync 预读完毕，再一次在 page cache 里寻找*/
        page = find_get_page(mapping, offset);
        if (!page)
            goto no_cached_page;
    }
    /*无论是否在 page cache 找到对应页还是重新启动了 readpage 都走到这里。走到这里的
      前提是获得了对应页，在使用前要 lock 该页*/
    if (!lock_page_or_retry(page, vma->vm_mm, vmf->flags)) {
        page_cache_release(page);
        return ret | VM_FAULT_RETRY;
    }
```

```
    //记录下找到的物理页，函数执行成功时从这里返回
    vmf->page = page;
    return ret | VM_FAULT_LOCKED;

no_cached_page:
    /*
     Page cache 里没有对应页，预读不允许或者也没有成功，这时启动 readpage 读取该页
     */
    error = page_cache_read(file, offset);

    /*
    读取完毕返回到 retry_find
     */
    if (error >= 0)
        goto retry_find;

…
}
```

6.1.2　File backed 内存的请页

本节分析 file backed 内存在其 pte 为 0 时发生的页异常的处理，这种情况发生的条件为：

（1）file backed 内存在首次被访问时，这时其只有 struct vm_area_struct 描述其属性，页表项还为空的状态。

（2）file backed 内存在被 shrink 掉的时候，但是 VM_SHARED 类型且被修改后的 page 不属于这种情况。

```
/*该函数的调用在系统运行时最常见，大量见于代码段访问以及 mapped 数据文件读访问等*/
static int __do_fault(struct mm_struct *mm, struct vm_area_struct *vma,
        unsigned long address, pmd_t *pmd,
        pgoff_t pgoff, unsigned int flags, pte_t orig_pte)
{
    …
    //异常地址做页对齐
    vmf.virtual_address = (void __user *)(address & PAGE_MASK);
    …
    /*该 page 对应磁盘上的文件，所以该 page 位置应该在该文件的 page cache 树里，这里
      使用上节描述的 File backed 虚拟内存段操作函数读取对应页*/
    ret = vma->vm_ops->fault(vma, &vmf);
    …
    /*
```

```
该异常是在写操作时发生的。要检查该虚拟地址段的属性，若其有 VM_SHARED 标志，将该操
作交给具体的文件系统。否则，一个 COW 发生了
 */
page = vmf.page;
if (flags & FAULT_FLAG_WRITE) {
    if (!(vma->vm_flags & VM_SHARED)) {
    /*在一个非共享的虚拟地址空间发生了写操作异常，说明 COW 发生了，这里要新分配一
      个匿名页，并把原有 page 的内容复制到这个新的匿名页中，这里发生的时机是：某进
      程 map 了自己的某段数据空间，当前该进程是第一次写访问该段数据空间，或者这个
      数据空间对应的 page 虽然被读过，但之后由于长时间不用被 shrink_page_list
      给释放掉了，所以该进程发生 pte 为 0 时的异常*/
`       anon = 1;
        //为当前进程分配一个物理页，这是一个匿名页
        page = alloc_page_vma(GFP_HIGHUSER_MOVABLE,
                    vma, address);
        …
        charged = 1;
        //把原有内容复印到新的物理页
        copy_user_highpage(page, vmf.page, address, vma);
        __SetPageUptodate(page);
    } else {
        /*VM_SHARED 类型的写访问，只在第一次访问时走到这里，由具体的文件系统来
          处理。主要工作是同步该 page 的 writeback，且将该页 page 结构和对应的
          pte 项置脏，参见 do_wp_page 的分析*/
        if (vma->vm_ops->page_mkwrite) {
            …
            tmp = vma->vm_ops->page_mkwrite(vma, &vmf);
            …
            page_mkwrite = 1;
        }
    }
}
//索引到页表项
page_table = pte_offset_map_lock(mm, pmd, address, &ptl);

/* 首先检查在读取该 page 时，页表项是否发生改动。这种情况发生在从异常跑到这里的时
   候，同一进程的别的线程也访问了该地址*/
if (likely(pte_same(*page_table, orig_pte))) {
    /* 页表项没有发生改动，没有别的线程先于我们*/
    //创建页表项
    entry = mk_pte(page, vma->vm_page_prot);
    //写操作发生对页表项置 dirty 和 write
    if (flags & FAULT_FLAG_WRITE)
        entry = maybe_mkwrite(pte_mkdirty(entry), vma);
    //该页为新分配的匿名页
```

```
        if (anon) {
            /*匿名页，对 COW，这里将该 page 加入到 struct vm_area_struct 的 struct
              anon_vma *anon_vma 列表中，从此该进程的这页数据就成为了匿名页*/
            page_add_new_anon_rmap(page, vma, address);
        } else {
            //该 page 被 map 到某进程虚拟地空间，该页的 map 计数增加
            page_add_file_rmap(page);
            if (flags & FAULT_FLAG_WRITE) {
                //写一个文件的内容，脏页产生
                dirty_page = page;
                get_page(dirty_page);
            }
        }
        //设置页表项
        set_pte_at(mm, address, page_table, entry);

        /*页表项更新了，根据具体架构来决定什么时候需要手动更新 cache */
        update_mmu_cache(vma, address, page_table);
    } else {
        //别的线程先于我们完成
    }
…
out:
    if (dirty_page) {
        //脏页的处理
        struct address_space *mapping = page->mapping;
        /*将脏页提交到文件系统，有可能将该文件对应设备回写线程的唤醒，参见相关章节*/
        if (set_page_dirty(dirty_page))
            page_mkwrite = 1;
        unlock_page(dirty_page);
        put_page(dirty_page);
        /*若该页之前就被标志脏页，则检查对应设备脏页是否到达写回条件，若条件满足启动
          写回机制，参见相关章节*/
        if (page_mkwrite && mapping) {
            balance_dirty_pages_ratelimited(mapping);
        }
        …
    } else {
        unlock_page(vmf.page);
        //存放原始数据的页，无用了，释放掉
        if (anon)
            page_cache_release(vmf.page);
    }

    …
}
```

6.1.3 匿名内存的请页

匿名内存的请页有两种情况如下：

（1）首次访问。zero page 的情况在这里处理，即防止分配出来的 page 携带别的进程的原有内容。MMU 隔离了进程间的地址，如果在这里出现跨进程内存访问的问题就比较搞笑了。

（2）匿名内存位于 SWAP 分区或文件。这种情况类似于 File backed 内存页被释放的情形，从 SWAP 分区或文件里读出对应的 page，再挂到 pte 上。因为这种情况导致页面出现经常的“脏”状态。而嵌入式或者手机系统使用的 emmc、sd card 存储介质都是 nand，频繁的写导致坏块的产生。所以在嵌入式或者手机系统通常不支持 SWAP 分区。本书也不讨论这方面的内容。

```
/* 匿名页的处理 */
static int do_anonymous_page(struct mm_struct *mm, struct vm_area_struct
*vma,
        unsigned long address, pte_t *page_table, pmd_t *pmd,
        unsigned int flags)
{
    …
    /* 匿名页的首次读，匿名页是新分配的内存，首次读在逻辑上是没有意义的，所以内核为其
       准备了一个特殊的 0 页*/
    if (!(flags & FAULT_FLAG_WRITE)) {
        //直接用 0 页作为其访问的目的
        entry = pte_mkspecial(pfn_pte(my_zero_pfn(address),
                        vma->vm_page_prot));
        //页表项
        page_table = pte_offset_map_lock(mm, pmd, address, &ptl);
        …
        //直接将 0 设置进其 pte
        goto setpte;
    }

    /*首次写匿名页，自然要为其分配一个物理页面，但是该页面可能携带了其他进程的信息，所
      以这里不仅要分配新的页面，而且需要将该页面清 0*/
    …
    page = alloc_zeroed_user_highpage_movable(vma, address);
    if (!page)
        goto oom;
    __SetPageUptodate(page);

    …
    //新的 pte 值
    entry = mk_pte(page, vma->vm_page_prot);
```

```
    if (vma->vm_flags & VM_WRITE)
        entry = pte_mkwrite(pte_mkdirty(entry));
    //pte 指针
    page_table = pte_offset_map_lock(mm, pmd, address, &ptl);
    …
    //记录将该页面反向映射
    page_add_new_anon_rmap(page, vma, address);

    //设置 pte
setpte:
    set_pte_at(mm, address, page_table, entry);
    …
}
```

6.1.4 COW 访问

COW 的访问有以下几种。

（1）上文提到的二进制镜像的初始化数据区，这种访问产生时由其 struct vm_area_struct 相应异常处理函数 int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);来处理，在该访问发生后该 page 内存空间就变成了匿名页。

（2）该进程 Fork 出新的进程再次访问该段数据区，就变成了页表项被置为保护状态的写异常。

（3）新进程若访问二进制镜像的初始化数据区未被老进程修改，则其仍满足第一种情况，仍由 struct vm_area_struct 相应异常处理函数 int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);来处理。

（4）新进程若访问老进程运行时新产生的匿名数据区，属于第二种情况。

（5）老进程 Fork 新进程后访问自己运行时产生的匿名数据区，因为老进程在 Fork 时自己的页表项也被置为写保护，所以将发生新进程同样的情况。

第二种情况的处理如下

```
/*
仅考虑该函数对 COW 的处理
*/
static int do_wp_page(struct mm_struct *mm, struct vm_area_struct *vma,
        unsigned long address, pte_t *page_table, pmd_t *pmd,
        spinlock_t *ptl, pte_t orig_pte)
    __releases(ptl)
{
    …
    //从页表项里找到锁定的 page
    old_page = vm_normal_page(vma, address, orig_pte);
    …

    if (PageAnon(old_page) && !PageKsm(old_page)) {
```

```
        …
        /*该 page 之前在 Fork 已经被写保护锁定，并被映射到不同的进程中，每映射一次其
          _mapcount 都被加一，以后每当进程访问一次该 page 对于的虚拟地址都会分配新的
          page 并将该 page 复制到新 page 中。而老的 page 都会从新访问进程里被 unmap
          掉，当最后一个进程访问时，就没有必要做复制工作了，继续使用该 page 即可*/
        if (reuse_swap_page(old_page)) {
            …
        /*pte 之前已经设置过，这里将该 page 的 struct address_space *mapping;指
          针指向当前 struct vm_area_struct 的 struct anon_vma *anon_vma;即可*/
            page_move_anon_rmap(old_page, vma, address);
            …
            goto reuse;
        }
       …
    } else if (unlikely((vma->vm_flags & (VM_WRITE|VM_SHARED)) ==
                    (VM_WRITE|VM_SHARED))) {

        …

reuse:
        …
        //将该页置为 yong
        entry = pte_mkyoung(orig_pte);
        //清除写保护
        entry = maybe_mkwrite(pte_mkdirty(entry), vma);
        //重新设置 pte
        if (ptep_set_access_flags(vma, address, page_table, entry,1))
            update_mmu_cache(vma, address, page_table);
        …
        //从这里返回
        if (!dirty_page)
            return ret;
        …
    }

    /*
    非最后一次访问 COW 区域
     */
    page_cache_get(old_page);
gotten:
    …
        //分配一个新的 page 用来存放 COW 数据
        new_page = alloc_page_vma(GFP_HIGHUSER_MOVABLE, vma, address);
        …
        //将 COW 数据复制过来
```

```
        cow_user_page(new_page, old_page, address, vma);
        …
    page_table = pte_offset_map_lock(mm, pmd, address, &ptl);
    if (likely(pte_same(*page_table, orig_pte))) {
        …
        //创建新的页表项
        entry = mk_pte(new_page, vma->vm_page_prot);
        entry = maybe_mkwrite(pte_mkdirty(entry), vma);
        …
        //每做一次 map 都记录下其反向映射
        page_add_new_anon_rmap(new_page, vma, address);
        /* 设置新的页表项 */
        set_pte_at_notify(mm, address, page_table, entry);
        …
        if (old_page) {
            /*old_page 已不在当前进程里，减少 old_page 的 map 计数*/
            page_remove_rmap(old_page);
        }


    } else
        mem_cgroup_uncharge_page(new_page);

    …
}
```

6.2　内存 Shrink

6.2.1　Shrink 操作 shrink_page_list

Linux 内核使用了尽可能多的内存，当新的内存要求来临时，将原有占用的不常用内存释放出去，挤出空间以满足新的内存需求。本节描述的就是这个挤占已用内存的框架函数。

```
/* shrink_page_list()主要工作步骤是拿到的页面链表 page_list 逐一检查，尽量释放 */
static unsigned long shrink_page_list(struct list_head *page_list,
                    struct zone *zone,
                    struct scan_control *sc)
{
    …
    //依次扫描 page_list 里的页面
    while (!list_empty(page_list)) {
```

```
        …
        //索引到该页面
        page = lru_to_page(page_list);
        //与 page_list 断开
        list_del(&page->lru);
        …
        /*若该页面是被 mlock 锁住的页面，即使再长时间没有被使用，也不能被释放掉，比如
          这是一个实时应用的页面，若被释放掉，再访问时需要启动磁盘操作，这是不可允许
          的。所以跳过这种类型的页面*/
        if (unlikely(!page_evictable(page, NULL)))
            goto cull_mlocked;
        …
        //若该页面处于 writeback 状态，根据情况等待其完成
        if (PageWriteback(page)) {
            //要看当前 shirnk 模式是否运行 sync 等待，否则直接跳过该页面
            if ((sc->reclaim_mode & RECLAIM_MODE_SYNC) &&
                may_enter_fs)
                //等待
                wait_on_page_writeback(page);
            else {
                unlock_page(page);
                //等不及，跳过该页面
                goto keep_lumpy;
            }
        }

        /*检查页面的引用状态：
(1) 若页面被映射到进程的虚拟地址空间，则依次索引其反向映射，并检查其对应的 pte 是否被置
位 L_PTE_YOUNG，若有一个 pte 项被置位 L_PTE_YOUNG，说明该页面新被访问，在清除其所有
反向映射 pte 的 L_PTE_YOUNG 后返回 PAGEREF_KEEP
(2) 若该页面 page 本身被置位 Referenced 则返回 PAGEREF_RECLAIM_CLEAN
(3) 若 (1)、(2) 同时发生则返回 PAGEREF_ACTIVATE
(4) 若无以上情况出现，返回 PAGEREF_RECLAIM
*/
  references = page_check_references(page, sc);
  //只要该页面新被访问，都得到保留，跳过此页
        switch (references) {
        case PAGEREF_ACTIVATE:
            //跳过
            goto activate_locked;
        case PAGEREF_KEEP:
            //跳过
    goto keep_locked;
        case PAGEREF_RECLAIM:
        case PAGEREF_RECLAIM_CLEAN:
```

```
        ; /*继续处理该页*/
    }

    /*匿名页且没有被加入 SWAP 的管理机制,这里将其加入,以便后面将该页 sync 到 SWAP
      后，再释放掉*/
    if (PageAnon(page) && !PageSwapCache(page)) {
        if (!(sc->gfp_mask & __GFP_IO))
            goto keep_locked;
        /* 实质上就是加入 SWAP 设备或文件的 mapping 里。即 struct address_space
           swapper_space 的 page tree 里*/
        if (!add_to_swap(page))
            goto activate_locked;
        may_enter_fs = 1;
    }

    mapping = page_mapping(page);

    /*
     反向映射的处理，主要是解开 MMU 的映射。见下文分析
     */
    if (page_mapped(page) && mapping) {
        switch (try_to_unmap(page, TTU_UNMAP)) {
        …
        case SWAP_AGAIN:
            //没有成功解开，跳过此页
            goto keep_locked;
        …
        case SWAP_SUCCESS:
            ; /*成功解开，继续处理此页 */
        }
    }
    /*脏页的处理，用 pageout 将其提交给 writeback 机制。这里有个 clean page 的
      动作，参见下文*/
    if (PageDirty(page)) {
        …
        /* 提交给 writeback 机制 */
        switch (pageout(page, mapping, sc)) {
        …
        case PAGE_CLEAN:
            ; /* 继续处理该页 */
        }
    }
```

```
            /* 该 page 有对应的 struct buffer_head，这是由 block 层产生的，用来映射部
               分 page 与磁盘地址的，这里将其释放掉*/
            if (page_has_private(page)) {
                //释放 struct buffer_head
                if (!try_to_release_page(page, sc->gfp_mask))
                    goto activate_locked;
                …
            }
            …
            //走到这里说明做好了对该 page 的释放工作
    free_it:
            nr_reclaimed++;
            //加入 free_pages，稍后将被释放
            list_add(&page->lru, &free_pages);
            continue;
            …
            //该页面不满足释放条件，继续保留
    activate_locked:
            …
    keep_lumpy:
            //加入 ret_pages 链表，该页将得到保留
            list_add(&page->lru, &ret_pages);
            …
        }
        …
    }
```

解映射，即将所有映射到某个 page 的 pte 都置 0。无论对于匿名页还是 file backed 页，做法都是扫描其反向映射链表，在对每个映射调用 try_to_unmap_one 解映射。

```
//解映射入口函数
int try_to_unmap(struct page *page, enum ttu_flags flags)
{
    int ret;
    …
    if (unlikely(PageKsm(page)))
        ret = try_to_unmap_ksm(page, flags);
    else if (PageAnon(page))
        //匿名页解映射
        ret = try_to_unmap_anon(page, flags);
    else
        //file backed 页的解映射
        ret = try_to_unmap_file(page, flags);
        …
    return ret;
```

```
}



/*
针对一个映射做解映射操作，这里重点关注 file backed 页的解映射
 */
int try_to_unmap_one(struct page *page, struct vm_area_struct *vma,
            unsigned long address, enum ttu_flags flags)
{
    …
    //找到相应的解映射
    pte = page_check_address(page, mm, address, &ptl, 0);
    …

    if (!(flags & TTU_IGNORE_ACCESS)) {
        //检查 pte 是否 L_PTE_YOUNG 且清除 L_PTE_YOUNG
        if (ptep_clear_flush_young_notify(vma, address, pte)) {
            ret = SWAP_FAIL;
            goto out_unmap;
        }
    }
    …
    /*这里记录下 pte 的值，并清除之，似乎是不经意一行代码，却解开该 page 与该 struct
      vm_area_struct *vma 之间的映射，断开了 struct vm_area_struct *vma 所在虚
      拟内存空间对该 page 的访问*/
    pteval = ptep_clear_flush_notify(vma, address, pte);
     //脏的处理
    if (pte_dirty(pteval))
        set_page_dirty(page);


    if (PageHWPoison(page) && !(flags & TTU_IGNORE_HWPOISON)) {
    //ARM 架构上不成立
    …
    } else if (PageAnon(page)) {
        /*如果是匿名页，且不支持 SWAP，则没有实质的处理，若支持 SWAP，则将 page 在 swap
          的位置放在 pte 里，本书重点分析 Android 操作系统下的内核行为，这里略过*/
        swp_entry_t entry = { .val = page_private(page) };
        …
        set_pte_at(mm, address, pte, swp_entry_to_pte(entry));
        …
    } …
…
}
```

6.2.2 Clean Page

在 shrink_page_list 将脏页提交到 block 层之前，内核需 clean 该 page。这包括两方面动作，一是 page 结构自身的 clean，另外一个就是该 page 对应的页表项的 clean，这个 clean 包含对这个页表项写保护 L_PTE_RDONLY 的重新置位。这样对于 map 到文件的内存在其下次被修改后能够产生写异常，以便内核能够监控脏页的产生。

```
/*在文件系统写回一个 page 之后，调用该函数 clean 该 page。事实上在虚拟文件系统层将脏写
  回时对于每个 page 也需要调用该函数*/
int clear_page_dirty_for_io(struct page *page)
{
    struct address_space *mapping = page_mapping(page);

    if (mapping && mapping_cap_account_dirty(mapping)) {
        //先做 page 结构本身的 clean，再进一步 clean 对应 pte
        if (page_mkclean(page))
            set_page_dirty(page);
    }
    return TestClearPageDirty(page);
}

//page MMU 映射方面的 clean 处理
int page_mkclean(struct page *page)
{
    int ret = 0;
    //若该 page 会映射到应用的虚拟内存
    if (page_mapped(page)) {
        struct address_space *mapping = page_mapping(page);
        if (mapping) {
            //进一步 clean 其 pte
            ret = page_mkclean_file(mapping, page);
            …
        }
    }
    …
}

/*一个 page 可能被映射到多个应用中，从反向映射链表中依次找出虚拟内存空间，并清除其对应
  pte*/
static int page_mkclean_file(struct address_space *mapping, struct page
*page)
{
    …
```

```
        //依次遍历映射该 page 的反向映射链表
        vma_prio_tree_foreach(vma, &iter, &mapping->i_mmap, pgoff, pgoff) {
            /*只有 VM_SHARED 才需要进行文件操作，才有做 clean 的必要*/
            if (vma->vm_flags & VM_SHARED) {
                unsigned long address = vma_address(page, vma);
                if (address == -EFAULT)
                    continue;
                //在一个虚拟地址空间做 pte 的 clean
                ret += page_mkclean_one(page, vma, address);
            }
        }
        …
    }

    //一个 pte 的 clean 操作，所谓 clean，重要的是设置其只读属性
    static int page_mkclean_one(struct page *page, struct vm_area_struct *vma,
                    unsigned long address)
    {
        …
        //找出 pte
        pte = page_check_address(page, mm, address, &ptl, 1);
        if (!pte)
            goto out;
        //pte 为脏或者 pte 没有置位，只读属性
        if (pte_dirty(*pte) || pte_write(*pte)) {
            pte_t entry;
            …
            //保留原先的 pte 内容后把该 pte 清 0
            entry = ptep_clear_flush_notify(vma, address, pte);
            //在原先的 pte 内容上置位 L_PTE_RDONLY
            entry = pte_wrprotect(entry);
            //清除原先的 pte 内容上的 L_PTE_DIRTY 标志
            entry = pte_mkclean(entry);
            //将处理好的 pte 写进页表
            set_pte_at(mm, address, pte, entry);
            ret = 1;
        }
        …
    }
```

6.2.3　脏页的监控

在 VM_WRITE|VM_SHARED 属性的 page 被 clean 之后，其 pte 为只读属性，若再次发生该 page 的写则产生新的写异常，走以下路径：

```
/*
只考虑该函数处理 clean 后且属性为 VM_WRITE|VM_SHARED 的 page 再次被写访问
*/
static int do_wp_page(struct mm_struct *mm, struct vm_area_struct *vma,
        unsigned long address, pte_t *page_table, pmd_t *pmd,
        spinlock_t *ptl, pte_t orig_pte)
    __releases(ptl)
{
    …
    //找出该 page
    old_page = vm_normal_page(vma, address, orig_pte);
    …
    if (PageAnon(old_page) && !PageKsm(old_page)) {
        //匿名页的情况不在这里讨论，参见上文
    } else if (unlikely((vma->vm_flags & (VM_WRITE|VM_SHARED)) ==
                    (VM_WRITE|VM_SHARED))) {
        /*int (*page_mkwrite)(…);主要作用是等待该页的 writeback 完成，若该页处于
          writeback 状态要等待其完成，才能再对其写操作，尽管内存操作仅仅是覆盖原先
          的内容，但是对于文件系统要满足其数据一致性的保护机制，所以这里需要交给具体
          的文件系统来做*/
        if (vma->vm_ops && vma->vm_ops->page_mkwrite) {
            …
            //由具体的文件系统来做两者的同步
            tmp = vma->vm_ops->page_mkwrite(vma, &vmf);
            …
            //做下标志，该 page 要使能写权限
            page_mkwrite = 1;
        }
        dirty_page = old_page;
        get_page(dirty_page);

reuse:
        …
        //该页新被访问，young 之
        entry = pte_mkyoung(orig_pte);
        //使能写
        entry = maybe_mkwrite(pte_mkdirty(entry), vma);
        //设置 pte
        if (ptep_set_access_flags(vma, address, page_table, entry,1))
            update_mmu_cache(vma, address, page_table);
        …
        //如果具体的文件系统没有提供 page_mkwrite 内核就需要自己做同步了
        if (!page_mkwrite) {
            //等待该页的 writeback
            wait_on_page_locked(dirty_page);
```

```
            /*将该页置脏，并启动写回机制，写回机制的启动不能太频繁，也不能太稀少*/
            set_page_dirty_balance(dirty_page, page_mkwrite);
        }
        put_page(dirty_page);
        //写发生标志
        if (page_mkwrite) {
            struct address_space *mapping = dirty_page->mapping;
            /*将该页置脏，为了支持不同的架构处理器，不同的文件系统，内核有些地方不得
              已有些重复操作，但这一般都是内存操作，不会损失性能*/
            set_page_dirty(dirty_page);
            …
            if (mapping)    {
                /*检查是否到了写回时机 */
                balance_dirty_pages_ratelimited(mapping);
            }
        }
        …
        /*下一步还要将该页的 pte 置脏位 L_PTE_DIRTY，这个工作在上一级函数完成。参见
          下文*/
        return ret;
    }
    …
}
```

6.3　全　景　图

本节分析缺页请页的框架函数，该函数由数据、指令转换异常而来，那里是虚拟内存触发时机的源泉。对于 Android 操作系统来说，这里才是内存管理真正开始的地方。

```
//页表项异常处理函数
int handle_pte_fault(struct mm_struct *mm,
            struct vm_area_struct *vma, unsigned long address,
            pte_t *pte, pmd_t *pmd, unsigned int flags)
{
    pte_t entry;
    spinlock_t *ptl;

    entry = *pte;
    /*pte 状态表明了其对应页面所处的状态*/
    if (!pte_present(entry)) {
        if (pte_none(entry)) {
        /*pte 为 0，表明有如下可能
File backed 或者匿名页第一次被访问
File backed 页被 shrink_page_list 释放掉之后，又一次被访问。参见 shrink_page_list
```

```
的分析，每次释放掉一个 File backed 页，其 PTE 都被置为 0。
*/
            if (vma->vm_ops) {
            /*vma->vm_ops 是在做内存的 file mmap 复制的，对于 File backed 页，该
              条件成立*/
                if (likely(vma->vm_ops->fault))
                    return do_linear_fault(mm, vma, address,
                        pte, pmd, flags, entry);
            }
            //第一次访问匿名页
            return do_anonymous_page(mm, vma, address,
                         pte, pmd, flags);
        }
        /*L_PTE_FILE 非线性的 file mmap，在 Android 系统里没有发现这种用法。这里是
          Linux 内核与系统关系非常好的实例。博大精深的 Linux 内核支持多种内存映射、
          文件读写、进程调度等方面的机制，但是操作系统往往不会使用其全部机制。一个好
          的操作系统，往往只是用到了其中一些适合自己上层建筑的机制。因为对于操作系统
          而言，一个良好的上层建筑是不可能把所有的 Linux 内核机制都利用到。读者也许会
          反问，ubuntu 之类的系统不是暴露了全部的 Linux 内核机制吗？ 但是笔者认为，
          ubuntu 之类的系统，只是 Linux 内核的外延，并不是一个完整的操作系统。而 Linux
          之所以在服务器上大行其道，是因为部分 redhat、ubuntu 所欠缺的操作系统功能
          被商业公司的中间件实现了。而且 ubuntu redhat 暴露的内核机制有多少会被这些
          承载操作系统机制的中间件使用到？再之，又有读者可能会反问：ubuntu 上面不是
          集成了 GTK、QT、等一堆组件，这些不是构成操作系统的要素吗？ 诚然，ubuntu
          上集成了一堆优秀的开源组件，但是其并不能构成操作系统。尽管 ubuntu 其上不乏
          大量优秀的开源组件，但是它们只是“堆”在那里，没有形成一个良好的架构，其之
          间没有有机的联系！且功能往往重复、互相之间没有留有合适的接口，没有良好的统
          一管理机制。这是开源 Linux 界的最大问题，各种项目各自为战，没有合力，无法形
          成良好的操作系统的上层建筑。为什么这种“攒”出来的 Linux 系统为什么一直在桌
          面、手机上一败再败，因为需要的是操作系统，提供的却只是内核和一堆不相干的组件
 */
        if (pte_file(entry))
            return do_nonlinear_fault(mm, vma, address,
                    pte, pmd, flags, entry);
        //匿名页，被 SWAP 到 SWAP 分区或文件中了
        return do_swap_page(mm, vma, address,
                    pte, pmd, flags, entry);
    }

    ptl = pte_lockptr(mm, pmd);
    spin_lock(ptl);
    if (unlikely(!pte_same(*pte, entry)))
        goto unlock;
    /*写异常。走到这里的情况不是因为野指针的写，那种情况在__do_page_fault 里就被截
      获了。这里的情况是 COW 的处理和脏页的监控*/
```

```
        if (flags & FAULT_FLAG_WRITE) {
            if (!pte_write(entry))
                return do_wp_page(mm, vma, address,
                        pte, pmd, ptl, entry);
            /*被 shrink 掉的 VM_SHARED 的非匿名内存走到这里，内核在这里监控脏页的产生*/
              entry = pte_mkdirty(entry);
        }
        //该页新被访问，置位 L_PTE_YOUNG
        entry = pte_mkyoung(entry);
        //设置 pte
        if (ptep_set_access_flags(vma, address, pte, entry, flags & FAULT_FLAG_
        WRITE)) {
            update_mmu_cache(vma, address, pte);
        } else {
            …
        }
    …
    }
```

第 7 章　块　设　备

在 Linux 体系下没有哪个驱动子系统的地位像 Block 那么重要。

整个数据流的最终搬运工作是由 Block 负责的。而存储介质的访问又是系统存储中最慢的地方。Block 的性能决定了整个系统性能表现。

对于不是基于块设备文件系统，文件系统基于一个特殊的驱动子系统处理存储介质，如 jffs2 直接架在 MTD 设备上。但是大部分的文件系统是基于块设备的，服务器、PC 使用 SCSI、SSD 自不用说。

在移动领域随着 emmc 的兴起，使其成为根目录文件系统已是不二选择，而 emmc 本身就是一种标准的 Block 类型设备，这使得智能手机根目录文件系统自然选择 ext4 了。

而对于非标准块设备，还可以通过驱动来模拟块设备，比如 nand 的设备，Linux 提供了一个 mtkblock block 驱动来模拟块设备，yaffs2 就基于这种设备接驳在 Block 上。

7.1　Bdev 文件系统

Bdev 文件系统是一个特殊的文件系统，它将块设备文件抽象为一个普通的 inode 节点，这样就可以使用普通文件处理框架来处理块设备，这样 Bdev 文件系统的节点就与系统中块设备对应起来。

其用途主要在两方面：一是直接使用原始块设备，就需要通过 Bdev 文件系统处理，Bdev 为其提供了像处理普通文件的 struct address_space_operations def_blk_aops；另一方面是被文件系统本身使用，文件系统在处理自己元数据时，实际上是直接对块设备进行处理，这时也通过 Bdev 文件系统处理来处理。

本节描述 Bdev 基本结构，但是对于 bdev 文件系统的使用才是其工作机制的关键，贯穿本章整个章节，参见下文。

Bdev 文件系统的描述如下：

```
static struct file_system_type bd_type = {
    .name        = "bdev",
    /*挂载时将执行该函数*/
    .mount       = bd_mount,
    .kill_sb     = kill_anon_super,
};
```

bdev 文件系统的挂载如下：

```
void __init bdev_cache_init(void)
{
```

```
    …
    //注册 bdev 文件系统
    err = register_filesystem(&bd_type);
    if (err)
        panic("Cannot register bdev pseudo-fs");
    //挂载 bdev 文件系统
    bd_mnt = kern_mount(&bd_type);
    …
}
/*bdev 是仅存在于内核的文件系统，适用伪文件系统挂载例程*/
static struct dentry *bd_mount(struct file_system_type *fs_type,
    int flags, const char *dev_name, void *data)
{
/*伪文件系统挂载：分配超级块、分配根目录节点*/
    return mount_pseudo(fs_type, "bdev:", &bdev_sops, NULL, 0x62646576);
}
/*bdev 超级块操作函数表*/
static const struct super_operations bdev_sops = {
    .statfs = simple_statfs,
    //分配新节点
    .alloc_inode = bdev_alloc_inode,
    …
};
```

bdev 文件系统的节点定义如下：

```
struct bdev_inode {
    struct block_device bdev;
    struct inode vfs_inode;
};
```

可见 bdev inode 是与 struct block_device bdev;紧密联系在一起的，而 bdev 文件系统的 inode 分配函数每次也是连同 struct block_device bdev;一起分配 bdev 的 inode 节点。

```
static struct inode *bdev_alloc_inode(struct super_block *sb)
{
   /* bdev_cachep 的单元大小就是 sizeof struct bdev_inode */
    struct bdev_inode *ei = kmem_cache_alloc(bdev_cachep, GFP_KERNEL);
    if (!ei)
        return NULL;
   /*以 struct  inode 的形式返回结果*/
    return &ei->vfs_inode;
}
```

为设备分配 bdev 文件系统的节点如下：

```
struct block_device *bdget(dev_t dev)
```

```
{
    struct block_device *bdev;
    struct inode *inode;
//参加上文 inode 的创建，这里调用到 bdev 文件系统的 inode 分配函数
    inode = iget5_locked(blockdev_superblock, hash(dev),
            bdev_test, bdev_set, &dev);

    if (!inode)
        return NULL;
// BDEV_I 返回结构 struct bdev_inode 的起始地址
    bdev = &BDEV_I(inode)->bdev;
//初始化 struct block_device 结构和 bdev 里的 struct inode 结构
    if (inode->i_state & I_NEW) {
        bdev->bd_contains = NULL;
// bd_inode 指向 bdev 的 inode
        bdev->bd_inode = inode;
        bdev->bd_block_size = (1 << inode->i_blkbits);
        …
        inode->i_bdev = bdev;
        //设置 bdev inode 的 i_data 默认操作
        inode->i_data.a_ops = &def_blk_aops;
        mapping_set_gfp_mask(&inode->i_data, GFP_USER);
        inode->i_data.backing_dev_info = &default_backing_dev_info;
        spin_lock(&bdev_lock);
        //挂入 all_bdevs 列表
        list_add(&bdev->bd_list, &all_bdevs);
        spin_unlock(&bdev_lock);
        unlock_new_inode(inode);
    }
    return bdev;
}
```

7.2 块设备基础结构

在分析块设备驱动之前，首先熟悉一下块设备的基本数据结构。

1. struct gendisk

内核通过块设备描述结构 struct gendisk 来描述整块块设备，代码如下：

```
struct gendisk {
    …
    int major;          /* major number of driver */
    int first_minor;
```

```
    //指示设备可以容纳的分区数目，如果==1 表示该设备不能被分区
    int minors;
    /*struct disk_part 包含着该块设备的分区数组，数组中每一项指向一个分区。而该分区
      数组的第 0 项比较特殊，它指向 struct gendisk 的成员变量 struct hd_struct
      part0;，代表块设备本身，当块设备没有分区时，分区数组只有这个唯一的一项*/
    struct disk_part_tbl __rcu *part_tbl;
    struct hd_struct part0;
    //块设备读写的请求队列
    struct request_queue *queue;
    …
}
```

2. struct hd_struct

struct hd_struct 描述块设备的其中一个分区。

```
struct hd_struct {
//起始扇区
    sector_t start_sect;
//该分区的大小
    sector_t nr_sects;
    …
//分区号
    int policy, partno;
    …
};
```

3. struct request_queue

struct request_queue 是块设备活动的中枢，记录了块设备的操作队列、电梯算法、请求操作触发函数等一系列关键信息。

```
struct request_queue {
    …
    struct list_head    queue_head;
    //上一次合并的操作
    struct request      *last_merge;
    //使用的电梯算法
    struct elevator_queue   *elevator;
    …
}
```

4. struct backing_dev_info 与 struct bdi_writeback

struct backing_dev_info 用来描述一个设备，包括块设备。该结构是主要用于内核回写机制：

```
struct backing_dev_info {
…
struct bdi_writeback wb;
…
}
```

“struct bdi_writeback”是回写控制结构，其工作原理如下

```
struct bdi_writeback {
    //回写 deamon
    struct task_struct *task;
    //回写时钟
    struct timer_list wakeup_timer;
    …
};
```

struct bdi_writeback 初始化时要为其设置定时回写时钟 struct timer_list wakeup_timer;，该时钟控制定时回写机制的周期性触发，参见回写部分。该时钟的触发函数为 wakeup_timer_fn：

```
static void wakeup_timer_fn(unsigned long data)
{
    struct backing_dev_info *bdi = (struct backing_dev_info *)data;

    spin_lock_bh(&bdi->wb_lock);
    if (bdi->wb.task) {
        //如果有定制的回写线程，叫醒之
        wake_up_process(bdi->wb.task);
    } else {
        //没有定制回写线程，使用内核默认的回写线程
        wake_up_process(default_backing_dev_info.wb.task);
    }
    spin_unlock_bh(&bdi->wb_lock);
}
```

7.3 块设备的创建与注册

整块块设备的注册是由块设备驱动发起的，首先块设备驱动要为其准备 struct block_device_operations *fops;和 struct request_queue *queue;结构。然后块设备驱动要调用 struct gendisk *alloc_disk(int minors)来为整块块设备创建其描述结构 struct gendisk 并注册该整块块设备。

```
//块设备注册函数
void add_disk(struct gendisk *disk)
```

```
{
    struct backing_dev_info *bdi;
    dev_t devt;
    int retval;
    …
    disk->flags |= GENHD_FL_UP;
    //part0 代表的是整块块设备，分配其设备号
    retval = blk_alloc_devt(&disk->part0, &devt);
    if (retval) {
        WARN_ON(1);
        return;
    }
    //part0 的 struct device __dev 代表了整块块设备
    disk_to_dev(disk)->devt = devt;

    //disk 的 major 为主设备号，first_minor 为第一个分区的 minor 设备号
    disk->major = MAJOR(devt);
    disk->first_minor = MINOR(devt);

    /* Register BDI before referencing it from bdev */
    bdi = &disk->queue->backing_dev_info;
    bdi_register_dev(bdi, disk_devt(disk));
    //向 bdev_map 注册该整块块设备的分区范围
    blk_register_region(disk_devt(disk), disk->minors, NULL,
                exact_match, exact_lock, disk);
    register_disk(disk);
    blk_register_queue(disk);

    …
}

/*整块块设备的注册，这里面最关键的一个动作是导致该整块块设备分区表的读取与分区结构的
  创建*/
void register_disk(struct gendisk *disk)
{
    struct device *ddev = disk_to_dev(disk);
    …

    /* 如果不支持分区，返回 */
    if (!disk_partitionable(disk))
        goto exit;

    /* 检查该设备是否存在，如果该设备存在必有容量值*/
    if (!get_capacity(disk))
        goto exit;
```

```
    /*part0 的 struct block_device，就是该整块块设备本身。这里将导致 bdev 文件系统
      为该整块块设备即 part0 创建节点*/
    bdev = bdget_disk(disk, 0);
    if (!bdev)
        goto exit;
    //bd_invalidated 置 1，将导致分区表的扫描
    bdev->bd_invalidated = 1;
    /*导致 static int __blkdev_get(…)的调用，见下文*/
    err = blkdev_get(bdev, FMODE_READ, NULL);
    …
}


/*void register_disk(struct gendisk *disk)导致该函数的调用，该函数在不同的调用
  时刻有着不同的路径*/
static int __blkdev_get(struct block_device *bdev, fmode_t mode, int
for_part)
{
    struct gendisk *disk;
    …
    //通过设备号找到对应的 struct gendisk 结构
    disk = get_gendisk(bdev->bd_dev, &partno);
    //在整块块设备注册时 bdev->bd_openers 不存在，因为这时没有人动过它
    if (!bdev->bd_openers) {
        bdev->bd_disk = disk;
        bdev->bd_contains = bdev;
        //对于整块块设备 partno 为 0
        if (!partno) {
            struct backing_dev_info *bdi;

            ret = -ENXIO;
            //part0 的 struct hd_struct
            bdev->bd_part = disk_get_part(disk, partno);
            if (!bdev->bd_part)
                goto out_clear;

            ret = 0;
            //如果整块块设备有打开函数，调用之，这取决具体的驱动实现
            if (disk->fops->open) {
                ret = disk->fops->open(bdev, mode);
                …
            }
            //如果没有人碰过这个块设备，要检查其 struct backing_dev_info
            if (!ret && !bdev->bd_openers) {
                bd_set_size(bdev,(loff_t)get_capacity(disk)<<9);
```

```
            bdi = blk_get_backing_dev_info(bdev);
            /*如果驱动层面没有提供 struct backing_dev_info，则使用默认的
              default_backing_dev_info 作为其 bdi，对于 emmc 这种情况不存在，
              emmc 驱动会提供自己的 struct backing_dev_info*/
            if (bdi == NULL)
                bdi = &default_backing_dev_info;
            bdev_inode_switch_bdi(bdev->bd_inode, bdi);
        }

        /*
         bdev->bd_invalidated 为 1，表示还没进行分区表的扫描，下一步要扫描分
         区表
         */
        if (bdev->bd_invalidated && (!ret || ret == -ENOMEDIUM))
            rescan_partitions(disk, bdev);
        if (ret)
            goto out_clear;
    } else {
        //如果打开的分区设备
        struct block_device *whole;
        //从 bdev 文件系统找出整块块设备的 struct block_device
        whole = bdget_disk(disk, 0);
        ret = -ENOMEM;
        //如果找不到对应的整块块设备，那么也没有必要下一步了
        if (!whole)
            goto out_clear;
        /*递归，去看一下整块块设备是否已经被打开、扫描过，如果没有，先要进行整块
          块设备的分区扫描*/
        ret = __blkdev_get(whole, mode, 1);
        if (ret)
            goto out_clear;
        //分区块设备的归属是整块块设备
        bdev->bd_contains = whole;
        /*找出分区块设备对应的 bdev 文件系统的 inode，并将其 inode->i_data.
          backing_dev_info 指向整块块设备的 bdi*/
        bdev_inode_switch_bdi(bdev->bd_inode,
            whole->bd_inode->i_data.backing_dev_info);
        //分区块设备的 struct hd_struct
        bdev->bd_part = disk_get_part(disk, partno);
        …
        //设置分区容量
        bd_set_size(bdev, (loff_t)bdev->bd_part->nr_sects << 9);
    }
} else {
    /* e2fsck 的时候走到这里，bd_openers 有值，且是打开的整块块设备，不过这个时
```

```
            候 bdev->bd_invalidated 已经被清 0，所以不会导致再次扫描分区表*/
        if (bdev->bd_contains == bdev) {
            ret = 0;
            if (bdev->bd_disk->fops->open)
                ret = bdev->bd_disk->fops->open(bdev, mode);
            /* the same as first opener case, read comment there */
            if (bdev->bd_invalidated && (!ret || ret == -ENOMEDIUM))
                rescan_partitions(bdev->bd_disk, bdev);
            if (ret)
                goto out_unlock_bdev;
        }
        /* only one opener holds refs to the module and disk */
        put_disk(disk);
        module_put(owner);
    }
    //成功被打开
    bdev->bd_openers++;
    …
}
```

7.4 分区检测生成

Linux 内核支持多种格式分区形式，针对每种分区格式，内核在 fs/partitions/check.c 里提供了检测函数：

```
/*分区格式检测函数列表。每种分区都有自己的检测函数，通过 CONFIG 来激活*/
static int (*check_part[])(struct parsed_partitions *) = {
…
#ifdef CONFIG_ACORN_PARTITION_ICS
    adfspart_check_ICS,
#endif
…
//这是常用的分区格式
#ifdef CONFIG_MSDOS_PARTITION
    msdos_partition,
#endif
}

/*分区监测与生成，设备注册时会触发该函数*/
int rescan_partitions(struct gendisk *disk, struct block_device *bdev)
{
    …
rescan:
    …
```

```
    //调用分区检测函数，state 返回的是监测到的分区数组
    if (!get_capacity(disk) || !(state = check_partition(disk, bdev)))
        return 0;

    …

    /* 针对检测到每一分区操作*/
    for (p = 1; p < state->limit; p++) {
        sector_t size, from;
        struct partition_meta_info *info = NULL;
        //记录分区大小
        size = state->parts[p].size;
        …
        //记录分区起点
        from = state->parts[p].from;

        /*向系统注册这个分区对应的块设备。该分区获得在内核中的身份 struct hd_struct*/
        part = add_partition(disk, p, from, size,
                    state->parts[p].flags,
                    &state->parts[p].info);
        …
    }
    …
}
```

7.5　块设备的打开

可以不考虑块设备作为 raw 设备被应用进程用 open 调用的情况，这种方式使用块设备在 Android 及大多数 PC 系统都是不存在的，仅考虑在加载基于块设备的文件系统时打开块设备的情况。

首先，这种方式打开的块设备有两个 struct inode 结构：

（1）struct inode 对应是该设备的设备文件，该 struct inode 位于这个设备文件所在文件系统，这是一个普通的 struct inode。

（2）struct inode 就是位于 bdev 文件系统，bdev 文件系统通过这个 struct inode 操作该设备。

Block 设备的 lookup 过程很好地说明了这个问题，代码如下：

```
struct block_device *lookup_bdev(const char *pathname)
{
    …
    /pathname 为块设备文件在当前文件系统里的位置，在当前文件系统查找它，查找的结果放
     在 path 里*/
```

```
    error = kern_path(pathname, LOOKUP_FOLLOW, &path);
    if (error)
        return ERR_PTR(error);
    /*从查找结果里取出设备文件在当前文件系统里的 inode*/
    inode = path.dentry->d_inode;
    error = -ENOTBLK;
    /*如果不是块设备，说明出错了*/
    if (!S_ISBLK(inode->i_mode))
        goto fail;
    error = -EACCES;
    /* path.mnt 指向当前文件系统，MNT_NODEV 表示不允许访问当前文件系统的特殊文件，
       如果这个条件成立，这个设备文件就不能打开*/
    if (path.mnt->mnt_flags & MNT_NODEV)
        goto fail;
    error = -ENOMEM;
/*将当前文件系统的 inode 与 struct block_device 联系起来*/
    bdev = bd_acquire(inode);
    …
}

static struct block_device *bd_acquire(struct inode *inode)
{
    struct block_device *bdev;
  /* bdev_lock 是保护块设备的锁*/
    spin_lock(&bdev_lock);
 /*inode 结构的 struct block_device *i_bdev;指针*/
    bdev = inode->i_bdev;
 /*如果指针不空表示已与 bdev 文件系统关联上*/
    if (bdev) {
        /*将 bdev inode 引用计数加一，表示又多了个使用者*/
        ihold(bdev->bd_inode);
        spin_unlock(&bdev_lock);
        return bdev;
    }
    spin_unlock(&bdev_lock);
 /*指针不空表示 bdev 文件系统对应的 inode 还没创建，调用 bdget 创建 bdev 文件系统的
   inode*/
    bdev = bdget(inode->i_rdev);
 /*bdev 为 bdev 文件系统的 inode 指针，参见 bdev 文件系统 inode 定义*/
    if (bdev) {
        spin_lock(&bdev_lock);
    /*建立其两个 inode 的联系*/
    if (!inode->i_bdev) {

            ihold(bdev->bd_inode);
```

```
            inode->i_bdev = bdev;
            /* i_mapping 指向 bdev 文件系统 inode 的 i_mapping ，说明如果有 open
               write，该 inode 有时也是通过 bdev 文件系统 inode 来处理*/
            inode->i_mapping = bdev->bd_inode->i_mapping;
    /*一个设备可能有多个设备文件，将这些设备文件串起来，挂在自己的 bd_inodes 链表上*/
            list_add(&inode->i_devices, &bdev->bd_inodes);
        }
        spin_unlock(&bdev_lock);
    }
    return bdev;
}
```

7.6　块设备驱动的层次结构

块设备驱动也分为多个层次，最下面的是芯片操作相关的驱动，如某个型号的 IDE、SCSI、MMC 控制芯片。再往上是协议相关的驱动子系统，如 IDE 类驱动、SCSI 类驱动、MMC 类驱动。再往上就是抽象的块设备驱动。若讨论具体的驱动子系统实现还可以再细分，但是针对 Linux 内核架构的讨论，分成三个层次足够了。本节仅讨论抽象的块设备驱动。而块设备驱动在内核架构的地位如此重要，为了能够从底到上、贯通的理解 Android 系统工作机理，本书另辟一章分析在手机、嵌入式领域大量使用的 EMMC、SD 卡块设备驱动。

抽象的块设备驱动位于文件系统的下侧和具体存储设备驱动的上侧。接受文件系统的提交的读写请求，并使用自身的电梯算法加以优化，再将请求下发给存储设备驱动。

文件系统将文件块的操作以 struct buffer_head 为单位提交到 block 层，block 层将其再次包装成 struct bio 提交到抽象块设备驱动。接着，抽象块设备驱动层找到对应块设备抽象出来的 struct request_queue 结构并执行其 make_request_fn*make_request_fn;函数。而下层的块设备驱动的需要根据自己的特殊情况来实现该函数。而对于块设备来说，有着很多共性，所以内核为其提供了 static int __make_request(…)例程，而大多数块设备子系统都接受了这个默认例程：

```
static int __make_request(struct request_queue *q, struct bio *bio)
{
    …
    /*
    Plug 机制，某个线程提交的块设备操作有着很强的连贯性，所以这里首先检查该 bio 是否能
    和当前线程 plug 里的 bio 合并
     */
    if (attempt_plug_merge(current, q, bio))
        goto out;

    spin_lock_irq(q->queue_lock);
```

```
    /*检查该 bio 是否可以和该设备待处理队列里的其他 bio 合并，elv_merge 指出是向前合
      并还是向后合并*/
    el_ret = elv_merge(q, &req, bio);
    if (el_ret == ELEVATOR_BACK_MERGE) {
        …
        //后向合并
        if (bio_attempt_back_merge(q, req, bio)) {
            if (!attempt_back_merge(q, req))
                elv_merged_request(q, req, el_ret);
            //后向合并成功，完成了该 bio 的处理
            goto out_unlock;
        }
    } else if (el_ret == ELEVATOR_FRONT_MERGE) {
         …
        //前向合并
        if (bio_attempt_front_merge(q, req, bio)) {
            if (!attempt_front_merge(q, req))
                elv_merged_request(q, req, el_ret);
            //前向合并成功，完成了该 bio 的处理
            goto out_unlock;
        }
    }

get_rq:
    //没有合并成功走到这里，检查该 bio 是读还是写
    rw_flags = bio_data_dir(bio);
    if (sync)
        rw_flags |= REQ_SYNC;

    /*
     下层的块设备驱动准备了一个 struct request_queue 数组，用来盛放抽象块设备驱动提
     交来的读写操作。这里即从 struct request_queue 数组里找到一项
     */
    req = get_request_wait(q, rw_flags, bio);

    /*
     把 bio 放到 struct request_queue 里
     */
    init_request_from_bio(req, bio);
…
    //检查 plug 机制
    plug = current->plug;
    if (plug) {
        /*一次读写操作可能会包含多个 bio，而这些 bio 之间往往存在连贯关系，所以对于一
          次读写，内核提供了 plug 机制，以便在整个读写操作都完成之后再启动底层的实际
```

```
        设备操作，以便进行 bio 之间的 merge。这里正是一次读写操作间的“bio”提交*/
        …
        //将这些 bio 串起来，待读写操作完成再 unplug 这个 bio 队列
        list_add_tail(&req->queuelist, &plug->list);
        …
    } else {
        …
        /*没有 plug，说明这个 bio 需要及时提交到下层驱动。struct request_queue 结
          构的 request_fn_proc *request_fn;函数是下层驱动实际工作的入口，通常由
          具体的设备子系统来实现。比如 static void mmc_request(struct request_
          queue *q)是 mmc 子系统的入口*/
        __blk_run_queue(q);
out_unlock:
        …
    }
…
}
```

7.7　虚拟块设备

有些存储介质的直接操作，尽管不具有块设备的特性。但是为了更好地适配到 Linux 内核的基础架构，在驱动层做了一层虚拟的块设备层，将对实际设备的操作都转换为基本的块设备操作。

以 Nand 的设备为例，本身上并不具备块设备特性，但是其上的文件系统如 Yaffs2 往往又需要使用块设备特性来接驳 Linux 内核基础架构。所以 Yaffs2 文件系统使用了虚拟的 Nandblock 设备。

```
/*文件系统加载基本操作：获取对块设备使用权，以 Yaffs2 为例分析*/
struct block_device *lookup_bdev(const char *pathname)
{
    …
/*查找该设备文件，path.dentry->d_inode 返回 rootfs 里对应于该设备文件的 inode*/
    error = kern_path(pathname, LOOKUP_FOLLOW, &path);
    …
//dev/block/mtdblock0 是块设备。不同于 jffs2 直接使用 mtd，yaffs2 把 nand 当成
  mtdblock 来用，这样给内核暴露出来的是一个标准的 block device，其暴露在 VFS 层面的
  行为特性就跟 ext3/4 很相似了
*/
    if (!S_ISBLK(inode->i_mode))
        goto fail;
    error = -EACCES;
    …
    //在 bdev 虚拟文件系统里为/dev/block/mtdblock0 分配一个节点
```

```
bdev = bd_acquire(inode);
    …
}
```

而在 mtdblock 层，其对 nand 的操作被封装在一个标准的块设备中，代码如下：

```
/*mtdblock 的抽象函数在 mtkblock 初始化时被触发*/
int add_mtd_blktrans_dev(struct mtd_blktrans_dev *new)
{
    /*struct gendisk 的分配对应于整个虚拟 mtdblock 设备*/
    gd = alloc_disk(1 << tr->part_bits);
    …
    /*struct request_queue 的分配用来接驳 block 层，内核所有对 nand 的读写操作都
      被 static void mtd_blktrans_request(…)转换成标准块设备操作了*/
    new->rq = blk_init_queue(mtd_blktrans_request, &new->queue_lock);

    …
   //向内核扫描并注册分区设备
    add_disk(gd);
    …
}
```

以上仅以 nand 设备作为分析实例，事实上作为一种最佳接驳 VFS 的机制，基于 Linux 内核系统使用大量的虚拟块设备机制。如 KVM guest 用的 Virtio_blk，网络块设备用的 NBD 等。

第 8 章　VFS

VFS 是 Linux 内核基本组成部分。从功能实现角度来说，内核只做了两件事情：一是“攒”除进线程运行的机制；二是给进线程提供一个与外界交互的窗口。VFS 就是这个窗口。

VFS 分为以下四层。

（1）第一层是一个抽象框架，对应于系统调用，由 Linux 内核实现，包括文件的打开、读写。

（2）VFS 第二层对应于 struct file_operations，有两种选择：一是提供特定文件系统，该文件系统的实现完全任意，只要满足内核抽象的文件的打开、读写等操作即可；第二种选择的背景是，内核基于其自身的回写机制以及文件的 page cache 机制提供了一套基本的文件操作函数。具体的文件系统在该层使用这套函数即可。

VFS 第二层是实现内核深度裁剪的最佳位置。 从内核实现的角度来讲，具体的文件系统只要满足基本的文件操作就能挂载到内核上，而不需要脏页的回写机制以及文件的 page cache 机制。而对于深度嵌入式系统，如小型的 NOMMU 的系统，其上文件系统非常简单，甚至可有可无。脏页的回写机制以及文件的 page cache 机制对其来说是个累赘。所以在这里实现特定文件系统，绕过 VFS 更深的层次，就可以显著简化内核，就像一个卸掉了负载车厢的火车头一样。

对于 Linux 系统上的大部分文件系统都需要使用 Linux 优秀的 page cache、回写等机制，所以基本上都会选择使用 Linux 提供的例程来实现。

（3）VFS 的第三层对应于 struct address_space_operations，是针对文件系统的具体的磁盘映射、预读、脏页提交等操作更深一步抽象的文件操作。对于标准的块设备的文件系统，内核在这里也提供像第二层一样，提供了基本的操作函数，文件系统的实现可以选择使用这些内核例程或重新实现。而对于一些非标准的块设备文件系统如 jffs2 就无法使用内核例程，必须自己实现。

（4）VFS 第四层是文件磁盘布局的体现。这里每个文件系统都不同，只能由具体的文件系统提供，如 int ext4_get_block(…)。

另外 VFS 还包括文件系统的加载、根目录的实现、目录 cache、inode cache 等方面的内容，这些与架构关系不大，且已经有很多介绍资料，本书不再介绍。

8.1　根　目　录

系统中总有一个根目录，本节讨论这个根目录的起源。

首先，根目录也是目录，也必然属于某个文件系统。内核要为根目录文件系统准备好

基础架构。

```
void __init mnt_init(void)
{…
// sysfs 文件系统的初始化
    err = sysfs_init();
…
//初始化根目录文件系统，根目录文件系统基于 ramfs 架构
    init_rootfs();
//生成/
    init_mount_tree();
}

//根目录文件系统的载体——ramfs
int __init init_rootfs(void)
{
    int err;
    //ramfs 的 bdi
    err = bdi_init(&ramfs_backing_dev_info);
    …
    //注册 rootfs_fs_type，其实际工作都是基于 ramfs
    err = register_filesystem(&rootfs_fs_type);
    …

    return err;
}

//加载根目录文件系统
static void __init init_mount_tree(void)
{
//mount roofs，其实就是 ramfs
    mnt = do_kern_mount("rootfs", 0, "rootfs", NULL);
    …
//mount 上的 fs 的 root 被当成整个系统的/
    root.mnt = ns->root;
    root.dentry = ns->root->mnt_root;
//设置到进程的当前目录
    set_fs_pwd(current->fs, &root);
//设置到进程的 root
    set_fs_root(current->fs, &root);


}
```

在往后生成 init 的操作都是使用 CLONE_FS 标识。如果不重新使得这个目录真正成为“/”。

8.1.1　根目录文件系统——initramfs

如果系统使用 initramfs 机制，内核通过宏 rootfs_initcall(populate_rootfs);将 initramfs 初始化函数编译进初始化函数指针列表。int __init populate_rootfs(void)函数得以执行。

该函数做如下操作：

（1）解压 __initramfs_end ---__initramfs_start 之间的压缩文件。

（2）根据解压出的内容执行如下动作：

```
static __initdata int (*actions[])(void) = {
    [Start]      = do_start,
    [Collect]    = do_collect,
    [GotHeader]  = do_header,
    [SkipIt]     = do_skip,
    [GotName]    = do_name,
    [CopyFile]   = do_copy,
    [GotSymlink]     = do_symlink,
    [Reset]      = do_reset,
};
```

（3）这些操作实际上是调用具体文件系统的操作函数、创建目录或复制文件。

以 GotName 操作为例，代码如下：

```
static int __init do_name(void)
{…
    if (S_ISREG(mode)) {//常规文件
        …
        int openflags = O_WRONLY|O_CREAT;
            …
        //以创建方式打开该文件
            wfd = sys_open(collected, openflags, mode);
    } else if (S_ISDIR(mode)) {//目录
        //创建目录
        sys_mkdir(collected, mode);
    } else if (S_ISBLK(mode) || S_ISCHR(mode) ||
            S_ISFIFO(mode) || S_ISSOCK(mode)) {
        if (maybe_link() == 0) {
        //创建设备节点
            sys_mknod(collected, mode, rdev);
        …
        }
    }
    return 0;
}
```

然而隐藏在这之后一个因素是，这些文件操作使用恰恰正是 ramfs 文件系统。在前面

的 static void __init init_mount_tree(void)里已经把文件系统的架子都搭好了，这里做的其实就是填充动作。

8.1.2 Android ramdisk.img

对于 Android，roofs 就是 ramdisk.img。Android 使用 ramfs 来构架它的最顶层的文件系统。不同于直接加载根目录设备，再找 init 的做法，Android 这种文件系统的构建方式给了 init 进程极大的灵活性，使其在完成系统初始化过程中自主加载主力文件系统。作为内核与系统的桥梁，init 进程可以看作内核的延伸，使其与内核一同加载，紧接其后运行，避免了内核必须过早的加载根目录设备驱动的生硬做法。从早期的 ramdisk 演进到 ramfs，在 Android 系统上得到了完美的体现。

（1）内核解压 initramfs，生成基本的最上层目录结构。

（2）Uboot 把 ramdisk.img 甩到内存的一个地址，并通知内核。

（3）内核启动到最后，找到 ramdisk.img，并把它解压并生成一个基于 cache 的文件系统——这就是 Android 最顶层的文件系统，init 进程存放的地方。

（4）内核释放掉 ramdisk.img 所占空间（因为 ramdisk.img 内容已经被取出构建 rootfs）。

（5）内核进入用户态，执行/init。

8.1.3 传统根目录文件系统加载方式

如果没有使用 initramfs，kernel_init 进入到 void __init prepare_namespace(void);以传统方式加载根目录文件系统。

（1）找到 root 设备。

（2）用已注册的文件系统去尝试加载该 root 设备到目录/root。

```
static int __init do_mount_root(char *name, char *fs, int flags, void *data)
{   //挂到/root
    int err = sys_mount(name, "/root", fs, flags, data);
    …
    //设置/root 为当前目录：set_fs_pwd(current->fs, &path);
    sys_chdir("/root");
    …
    return 0;
}
```

（3）把当前目录 move 到/，设置当前目录到 FS 的 root（set_fs_root(current->fs, &path);）。

8.2 文 件 打 开

文件是进程的窗户。

8.2.1 目录的层级查找

目录也是文件，只是里面的内容是描述的文件系统结构。目录的查找就是在这些目录文件中找到需要的描述项。

```
/*
该函数逐层解析文件路径，并在上层路径分量里的 dcache 或者文件系统里，查找下一层路径分量，
直到查找到最终节点
 */
static int link_path_walk(const char *name, struct nameidata *nd)
{
    …
//跳过连续的/
    while (*name=='/')
        name++;
    if (!*name)
        return 0;

    /* 逐层查找*/
    for(;;) {
        unsigned long hash;
        struct qstr this;
        unsigned int c;
        int type;

        nd->flags |= LOOKUP_CONTINUE;

        err = may_lookup(nd);
        if (err)
            break;
        //this 指向当前分量
        this.name = name;
        c = * (const unsigned char *)name;

        hash = init_name_hash();
        //计算当前分量 hash 值，dcache 里组织是以文件名为基础的
        do {
            name++;
            hash = partial_name_hash(c, hash);
            c = * (const unsigned char *)name;
        } while (c && (c != '/'));
        //计算当前分量的长度
        this.len = name - (const char *) this.name;
        //当前分量 hash 值，用来在 dcache 里查找
```

```
            this.hash = end_name_hash(hash);

            type = LAST_NORM;
            //处理“..”、“.”的情况
            if (this.name[0] == '.') switch (this.len) {
                case 2:
                    if (this.name[1] == '.') {
                        type = LAST_DOTDOT;
                        nd->flags |= LOOKUP_JUMPED;
                    }
                    break;
                case 1:
                    type = LAST_DOT;
            }
            …
            //在目录 nd->path.dentry 查找 this，nd->inode 将记录下查找结果的 inode
            err = walk_component(nd, &next, &this, type, LOOKUP_FOLLOW);
            …
            //如果没有 lookup 函数，说明该文件不是目录文件，(break)
            if (!nd->inode->i_op->lookup)
                break;
            continue;

    last_component:
            /* 清楚 LOOKUP_CONTINUE 标志，表示查找任务完成 */
            nd->flags &= lookup_flags | ~LOOKUP_CONTINUE;
            nd->last = this;
            nd->last_type = type;
            return 0;
        }
        …
    }


    static int do_lookup(struct nameidata *nd, struct qstr *name,
                struct path *path, struct inode **inode)
    {
        struct vfsmount *mnt = nd->path.mnt;
        /* nd->path.dentry 指向父目录的 struct dentry 结构，该函数的目的是在该目录下
           查找 struct qstr *name 文件*/
        struct dentry *dentry, *parent = nd->path.dentry;
        int need_reval = 1;
        int status = 1;
        int err=0;
       …
```

```
    //先在 dcache 里查找
                dentry = __d_lookup(parent, name);
…
//没有在 dcache 里找到，去文件系统里找
    if (unlikely(!dentry)) {
        //父目录也是一个 inode，取出父目录的 inode
        struct inode *dir = parent->d_inode;
        …

        mutex_lock(&dir->i_mutex);
        //锁住该目录后再查一次 dacahe
        dentry = d_lookup(parent, name);
        //依然没有找到
        if (likely(!dentry)) {
            /*这里再为当前文件 name 分配一个新的 struct dentry 并将其挂载到父目录
              的 struct list_head d_subdirs;链表里*/
            dentry = d_alloc_and_lookup(parent, name, nd);
            …
            if (IS_ERR(dentry)) {
            //如果没有成功查找，解开当前目录 mutex 锁，返回错误值
                mutex_unlock(&dir->i_mutex);
                err = PTR_ERR(dentry);
                goto end_do_lookup;
            }
            /* need_reval 为 0 表示不需要更新 dcache*/
            need_reval = 0;
            //status 为 1，表示成功
            status = 1;
        }
        //成功查找，解开当前目录 mutex 锁
        mutex_unlock(&dir->i_mutex);
    }
    …
    //将查找结果记录下来
    path->mnt = mnt;
    path->dentry = dentry;
    err = follow_managed(path, nd->flags);
    …
    *inode = path->dentry->d_inode;
    …
}


static struct dentry *d_alloc_and_lookup(struct dentry *parent,
                struct qstr *name, struct nameidata *nd)
```

```
{
    struct inode *inode = parent->d_inode;
    struct dentry *dentry;
    struct dentry *old;
    …
    /* 从 dcache 里分配一个新的 struct dentry*/
    dentry = d_alloc(parent, name);
    …
    /* inode 为父目录节点，从里面查找 dentry，对于 ext4，目录节点的操作函数为： struct
       inode_operations ext4_dir_inode_operations，所以这里导致 ext4_lookup
       函数的调用*/
    old = inode->i_op->lookup(inode, dentry, nd);
    if (unlikely(old)) {
        dput(dentry);
        dentry = old;
    }
    return dentry;
}
```

下面是具体文件系统的文件查找，以 ext4 文件系统为例，Ext4 inode 节点的查找有以下步骤。

（1）查找所在目录的目录项。

（2）根据目录项描述的节点号，取出 inode。

```
static struct dentry *ext4_lookup(struct inode *dir, struct dentry *dentry,
struct nameidata *nd)
{
    struct inode *inode;
    struct ext4_dir_entry_2 *de;
    struct buffer_head *bh;
…
    /*在 ext4 的 dir 目录下查找 dentry 对应的文件，返回的是该文件在 dir 目录下的目录项*/
    bh = ext4_find_entry(dir, &dentry->d_name, &de);
    inode = NULL;
    if (bh) {
        //de 即为待查文件的目录项，从目录项里取得该文件在 ext4 里的文件号
        __u32 ino = le32_to_cpu(de->inode);
        …
        //取出 ext4 的节点
        inode = ext4_iget(dir->i_sb, ino);
        …
    }
    return d_splice_alias(inode, dentry);
}
```

8.2.2　各层次操作函数的安装

VFS 的层次设计，允许具体的文件系统实现在每一层既可以实现替换默认操作的函数，也可以使用默认操作而完成其逻辑填空即可。在文件打开的过程，要为文件每一层配置相关操作函数。

第二层文件操作函数，该层对应于进程文件：

```
struct file_operations {
    struct module *owner;
    //文件操作指针定位
    loff_t (*llseek) (struct file *, loff_t, int);
    //文件读
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    //文件写
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    //文件异步读
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long,
    loff_t);
    //文件异步写
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long,
    loff_t);
    //读目录文件
    int (*readdir) (struct file *, void *, filldir_t);
    …
    //文件 map 操作，接驳 VM
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    …
};
```

VFS 第三层操作函数，该层对应于物理文件节点，有两个层面。

Inode 层面操作：

```
struct inode_operations {
    //查找目录，目录文件节点须实现该操作
    struct dentry * (*lookup) (struct inode *,struct dentry *, struct
    nameidata *);
    //跟进链接文件，打开文件时用到
    void * (*follow_link) (struct dentry *, struct nameidata *);
    …
    //文件创建
    int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
    //删除文件时使用
    void (*truncate) (struct inode *);
```

```
    //设置属性
    int (*setattr) (struct dentry *, struct iattr *);
    //获取属性
    int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
    ...
}
```

初看起来，inode 操作列表没有为文件操作提供对应的底层服务，其原因在与 inode 类似于一个容器其成员变量 struct address_space i_data;就是 page cache 树，所有文件操作都被转化为对 struct address_space 的操作：

```
struct address_space_operations {
    /*写一页，产生 Block 层提交，严格地来说 writepage 并不属于 VFS 第三层，该函数主要
      是提供给脏页回写机制以及页面收缩机制使用*/
    int (*writepage)(struct page *page, struct writeback_control *wbc);
    //读一页，产生 Block 层提交
    int (*readpage)(struct file *, struct page *);
    ...
    /*值得一提是 direct_IO，即使是对于 ext4 这些使用 Page cache 机制的文件系统，依然
      可以通过 direct_IO 操作来避开 page cache，这样所有的文件内容将不会被加到 page
      cache 里*/
    ssize_t (*direct_IO)(int, struct kiocb *, const struct iovec *iov,
            loff_t offset, unsigned long nr_segs);
    ...

};
```

VFS 在各个层面为每种文件操作都提供了自己的函数，VFS 提供的函数完美地实现了上文提到 cache 机制，事实上 Linux 体系下的主流文件系统都使用了 VFS 提供的函数，这里面最典型的例子就是 Ext4。不过，尽管从代码树上看属于 VFS 的例程，但是逻辑上，这些 VFS 提供的函数是特定文件系统的一部分。

8.3 文 件 写

8.3.1 文件写框架

VFS 将文件写分为以下 4 个基本动作。

（1）找出待写入文件内容对应的逻辑块。

（2）将文件内容取出并放入对应的 page cache。

（3）将文件写内容对应的 page 标志为脏，且将文件节点 inode 标志为脏，并将该节点挂入其超级块的待写回队列。

（4）检查内存里脏页是否到达一定阈值，如果满足条件则唤醒写回 deamon。

VFS 提供了一个基础的写框架的例程 static ssize_t generic_perform_write(…)，代码如下：

```
//struct iov_iter *i 里存放着要写入的文件内容，pos 则为文件指针
static ssize_t generic_perform_write(struct file *file,
                struct iov_iter *i, loff_t pos)
{
    …
    do {
    …
again:
        /*write_begin 用来映射文件逻辑块或者分配文件块，同时将该 page 对应的文件内
        容读进内存*/
        status = a_ops->write_begin(file, mapping, pos, bytes, flags,
                        &page, &fsdata);
        if (unlikely(status))
            break;
        /*write_begin 会把文件内容读取出来，这种情况下有可能进行 DMA 操作导致与 L1
          dcache 不一致，所以要刷一下 L1 dcache，以保持一致性，如果其内容有可能为可
          执行代码，还要刷一下 L1 icache*/
        if (mapping_writably_mapped(mapping))
            flush_dcache_page(page);

        pagefault_disable();
        /*把文件内容从用户态读进内核的 page 里*/
        copied = iov_iter_copy_from_user_atomic(page, i, offset, bytes);
        pagefault_enable();
        /*再次刷 L1 cache，这次刷 cache 的原因不同于上一次，这里在于在把文件内容从用
          户态读进内核的 page 里的过程中，使用虚拟地址对 page 进行了写操作，这时该 page
          的内容有可能还在 dcache 里，而这之后有可能对该 page 做 DMA，所以这里也要刷
          一下*/
        flush_dcache_page(page);
        /*针对所有的 page cache 里的 page，内核将其按其活跃度串在不同的“活跃”或“不
          活跃”队列，在 shrink memory 时作为判断标准，这里说明该 page 活跃度增加，
          改变其活跃状态*/
        mark_page_accessed(page);
        /*写操作的结束阶段，写操作的结束，不等于真正写回存储设备，这里只是将其标志
          为脏*/
        status = a_ops->write_end(file, mapping, pos, bytes, copied,
                        page, fsdata);
        …
        //文件指针迁移
        pos += copied;
        written += copied;
        //检查是否到了写回时机
```

```
        balance_dirty_pages_ratelimited(mapping);
    //步进 struct iov_iter 下一个内容段
    } while (iov_iter_count(i));
    //返回写入的长度，其实是写入到 page cahe 里的长度
    return written ? written : status;
}
```

8.3.2 write_begin

这是写操作的第一个阶段，为要写入的文件内容映射或者分配文件块。每种文件系统根据自己的特点实现各有不同，但是几乎所有的块设备文件系统都使用 int __block_write_begin(…)作为其 write_begin 的主体。

```
//不同的文件系统都需要提供自己的 get_block_t *get_block 函数
int __block_write_begin(struct page *page, loff_t pos, unsigned len,
        get_block_t *get_block)
{
    //写起始地址，这里以页为单位
    unsigned from = pos & (PAGE_CACHE_SIZE-1);
    //写终止地址
    unsigned to = from + len;
    …

    //逻辑块的大小
    blocksize = 1 << inode->i_blkbits;
    //如果 page 没有 bh，要为其创建 bh，每个 bh 指向一个逻辑块
    if (!page_has_buffers(page))
        create_empty_buffers(page, blocksize, 0);
    head = page_buffers(page);

    bbits = inode->i_blkbits;
    //算出 page 对应的文件系统起始逻辑块号
    block = (sector_t)page->index << (PAGE_CACHE_SHIFT - bbits);

    //针对 page 对应的每一个逻辑块，映射或分配磁盘块
    for(bh = head, block_start = 0; bh != head || !block_start;
        block++, block_start=block_end, bh = bh->b_this_page) {
        block_end = block_start + blocksize;

        //如果要写的内容不在这个逻辑块中，跳过
        if (block_end <= from || block_start >= to) {
            if (PageUptodate(page)) {
                if (!buffer_uptodate(bh))
                    set_buffer_uptodate(bh);
            }
```

```
                continue;
            }
            if (buffer_new(bh))
                clear_buffer_new(bh);
            /*该 BH 没有映射过，说明其没有对应的磁盘地址，而其逻辑块对应哪个磁盘地址，只有
              具体文件的文件系统才知道*/
            if (!buffer_mapped(bh)) {
                //用具体的文件系统映射函数，取该逻辑块对应磁盘地址
                err = get_block(inode, block, bh, 1);
                if (err)
                    break;
            /*如果这是一个新分配的逻辑块，要在对应的设备 page cache 树里查找是否已经存在。
              出现这种可能性的原因在于，文件系统元数据使用的设备文件的 page cache 树，详
              细分析参见在 block 一章*/
                if (buffer_new(bh)) {
    //处理磁盘块同时挂载普通文件件 page cache 和设备文件 page cache 的情况
                    unmap_underlying_metadata(bh->b_bdev,
                                bh->b_blocknr);
                    …
                    /*如果写入内容落在该逻辑块，那么要把 page 里对应地方擦干净，因为某些
                      文件系统这时已经读进内容了，而该逻辑块又是新分配的，而这个逻辑块有
                      可能是别的用户的文件，为了安全性着想，清 0*/
                    if (block_end > to || block_start < from)
                        zero_user_segments(page,
                            to, block_end,
                            block_start, from);
                    continue;
                }
            }
            …
            //逻辑块映射完毕，且写目标落入该逻辑块，下一步要将逻辑块读出来
            if (!buffer_uptodate(bh) && !buffer_delay(bh) &&
                !buffer_unwritten(bh) &&
                (block_start < from || block_end > to)) {
                //提交逻辑块读申请
                ll_rw_block(READ, 1, &bh);
                //记录下需要等待的逻辑块
                *wait_bh++=bh;
            }
        }
        /*等待逻辑块读完毕再返回，因为下一步要往 page 里写东西，如果不等待 update，就会出错*/
        while(wait_bh > wait) {
            wait_on_buffer(*--wait_bh);
            if (!buffer_uptodate(*wait_bh))
                err = -EIO;
```

```
        }
        ...
    }
```

8.3.3 write_end

本节分析写框架的 write_end 例程，内核提供通用 write_end 例程，基于内核的 static int __block_commit_write 工作可被具体文件系统直接引用。

```
//write_end 例程主要工作是脏页及脏节点的提交
int generic_write_end(struct file *file, struct address_space *mapping,
            loff_t pos, unsigned len, unsigned copied,
            struct page *page, void *fsdata)
{
    struct inode *inode = mapping->host;
    int i_size_changed = 0;
    //将本次写覆盖范围内所有的页提交为脏页，参见下文
    copied = block_write_end(file, mapping, pos, len, copied, page, fsdata);

    //文件变长了，在 inode 里记录新的文件长度
    if (pos+copied > inode->i_size) {
        i_size_write(inode, pos+copied);
        i_size_changed = 1;
    }
    ...
    //inode 本身发生变化，将 inode 本身标志为脏
    if (i_size_changed)
        mark_inode_dirty(inode);
    return copied;
}
```

写进 page cache 而有没有被写回磁盘的 page 称之为脏页。文件写的最后一个动作是维护脏页，代码如下：

```
static int __block_commit_write(struct inode *inode, struct page *page,
        unsigned from, unsigned to)
{
    ...
    blocksize = 1 << inode->i_blkbits;
    /* 针对 page 的每一个 BH 做两个动作：（1）是否该 BH 已经 update，在 block 层驱动在
      BH read 完成以后会将该 BH 置为 UPADTE；（2）将该 BH 标志为脏，这将引起 inode 写
      回队列的变化，见下文详述*/
        for(bh = head = page_buffers(page), block_start = 0;
        bh != head || !block_start;
        block_start=block_end, bh = bh->b_this_page) {
```

```
            block_end = block_start + blocksize;
            if (block_end <= from || block_start >= to) {
            //该 BH 没有 update
                if (!buffer_uptodate(bh))
                    partial = 1;
            } else {
                set_buffer_uptodate(bh);
            //标志该 BH 为脏
                mark_buffer_dirty(bh);
            }
            clear_buffer_new(bh);
        }
        //有一逻辑块不为 upadate，也不能置为 update
        if (!partial)
            SetPageUptodate(page);
        return 0;
    }
```

8.4　脏页的提交与回写机制

8.4.1　脏页的提交

脏页的提交主要有以下两种原因。

（1）文件系统的写导致了脏页，在文件写的最后需提交脏页，这里还包括由于文件增长导致 inode 本身为脏的提交。

（2）VM_SHARED 属性的 File backed 内存发生了写访问。

文件写操作导致的脏页提交的代码如下：

```
//若一个逻辑块为 dirty，则整个 page 都是 dirty
void mark_buffer_dirty(struct buffer_head *bh)
{
    //已处于 dirty 状态，返回
    if (buffer_dirty(bh)) {
        smp_mb();
        if (buffer_dirty(bh))
            return;
    }
    //该 BH 第一次被置位为脏
    if (!test_set_buffer_dirty(bh)) {
        struct page *page = bh->b_page;
        //查看对应的 page 是否为脏
        if (!TestSetPageDirty(page)) {
            struct address_space *mapping = page_mapping(page);
```

```
                //该 page 有一个逻辑块为脏，那么该 page 就处于脏状态
                if (mapping)
                    __set_page_dirty(page, mapping, 0);
            }
        }
}

//一个文件里有一个 page 为脏，那么该文件也必定为脏
static void __set_page_dirty(struct page *page,
        struct address_space *mapping, int warn)
{
    …
    //将该文件节点标志为脏
    __mark_inode_dirty(mapping->host, I_DIRTY_PAGES);
}


//inode 节点脏使能
void __mark_inode_dirty(struct inode *inode, int flags)
{
    struct super_block *sb = inode->i_sb;
    struct backing_dev_info *bdi = NULL;

    /*inode 节点本身为脏，经常发生的情况是文件长度增加、属性改变*/
    if (flags & (I_DIRTY_SYNC | I_DIRTY_DATASYNC)) {
        //调用对应超级块的 inode 脏使能函数
        if (sb->s_op->dirty_inode)
            sb->s_op->dirty_inode(inode, flags);
    }


    /*如果 inode 状态与要设置的状态一致，这没有必要再进行了。一个 dirty 有多个 page，
      多个 BH，每一次都移动 inode 队列没有必要，而且正如注释所述，将引起不必要的
      inode->i_lock 竞争 */
    if ((inode->i_state & flags) == flags)
        return;


    //该 inode 状态与要更新的状态不一致
    spin_lock(&inode->i_lock);
    if ((inode->i_state & flags) != flags) {
        /*inode 有多种状态，这里检查是否已经处在 inode 节点脏或是数据页脏的情况，
          was_dirty 表示过去就是脏的*/

        const int was_dirty = inode->i_state & I_DIRTY;
```

```
            //置位
            inode->i_state |= flags;

            /*
            该节点正处在 SYNC 状态，跳过
             */
            if (inode->i_state & I_SYNC)
                goto out_unlock_inode;

            …
            //该节点第一次被置为脏
            if (!was_dirty) {
                bool wakeup_bdi = false;
                //索引到该节点所在的设备信息
                bdi = inode_to_bdi(inode);
                //检查该节点是否能写回，一般的块设备都具备
                if (bdi_cap_writeback_dirty(bdi)) {
                    …
                    /*设备的写回控制结构挂载着设备当前的脏节点和正在写回的 inode 节点，
                      以其是否为空来判断写回 deamon 的状态*/
                    if (!wb_has_dirty_io(&bdi->wb))
                        wakeup_bdi = true;
                }
                //脏使能关键的操作
                spin_unlock(&inode->i_lock);
                spin_lock(&inode_wb_list_lock);
                inode->dirtied_when = jiffies;
                //将该节点挂载到写回控制结构的脏队列
                list_move(&inode->i_wb_list, &bdi->wb.b_dirty);
                spin_unlock(&inode_wb_list_lock);
                //叫醒 bdi 的 deamon，参见 block 一章
                if (wakeup_bdi)
                    bdi_wakeup_thread_delayed(bdi);
                return;
            }
        }
    …

    }
```

8.4.2　回写时机

内核使用回写机制的目的是减少磁盘操作的次数，在一个 page 的修改累积到一定程度再启动底层物理设备写操作。回写机制的触发时机有以下三种。

（1）来自文件写。在文件写操作的结束，通常检查脏页的状态是否满足启动回写机制的条件。

（2）来自 File backed 且 VM_SHARED 的内存写。在页异常产生脏页的时候也会检查脏页的状态是否满足启动回写机制的条件，而在 shrink_page_list 时会直接将脏页用 writepage 提交到设备层。

（3）来自存储设备的定时机制。存储设备 struct backing_dev_info 在向内核内核注册自身时，会启动一个内核线程，该线程会定时醒来，检查并回写该设备的脏页。

除第二个触发时机的 shrink_page_list 直接 writepage 的情况以外，触发时机的到来不一定会一定触发回写机制，触发时机到来时系统中脏页需要满足以下条件才会进一步触发写操作的实际执行，参见如下代码：

```
//触发回写机制的条件检查
void balance_dirty_pages_ratelimited_nr(struct address_space *mapping,
                        unsigned long nr_pages_dirtied)
{
    …
    /*第一个条件，即触发时机对应的设备上的脏页累积超过一定数量*/
    if (mapping->backing_dev_info->dirty_exceeded)
        ratelimit = 8;

    /*
     第二个触发条件，当前处理器上产生的脏页是否超过系统限定值 ratelimit_pages
     */
    preempt_disable();
    //当前处理器上产生的脏页计数
    p =  &__get_cpu_var(bdp_ratelimits);
    //当前处理器上产生的脏页累加
    *p += nr_pages_dirtied;
    //大于阈值才会启动回写机制
    if (unlikely(*p >= ratelimit)) {
        …
        balance_dirty_pages(mapping, ratelimit);
        return;
    }
    …
}
```

第三个触发时机保证了在满足不了上文提到的回写触发条件的脏页回写。在一个脏页产生数量较少的情况，上一个检查机制失效，只有通过定时机制来保证其回写了，参见如下代码：

```
/*每个 struct backing_dev_info 在注册时都会创建以下内核线程，以检测需要回写到该设备
  的脏页状态*/
static int bdi_forker_thread(void *ptr)
```

```
{
    //me 即为当前 struct backing_dev_info 的回写控制机构
    struct bdi_writeback *me = ptr;
    …
    for (;;) {
        …
        //action 是针对当前脏页状态需要做出的相关动作
        enum {
            NO_ACTION,   /* Nothing to do */
            FORK_THREAD, /* Fork bdi thread */
            KILL_THREAD, /* Kill inactive bdi thread */
        } action = NO_ACTION;
        /*
         当前设备有脏页待写，则回写待回写脏页
         */
        if (wb_has_dirty_io(me) || !list_empty(&me->bdi->work_list)) {
            del_timer(&me->wakeup_timer);
            wb_do_writeback(me, 0);
        }
        …
        /*检查系统中的其他 struct backing_dev_info 设备，是否也有待回写脏页*/
        list_for_each_entry(bdi, &bdi_list, bdi_list) {
            bool have_dirty_io;
            …
            //该 struct backing_dev_info 设备存在待回写脏页
            have_dirty_io = !list_empty(&bdi->work_list) ||
                    wb_has_dirty_io(&bdi->wb);

            //但是该 struct backing_dev_info 设备上却没有活跃的回写线程
            if (!bdi->wb.task && have_dirty_io) {
                …
                /*action 指出需要在该设备激活回写 deamon*/
                action = FORK_THREAD;
                break;
            }
            ..
            //该设备已无待回写脏页，但是其上还有回写 deamon
            if (bdi->wb.task && !have_dirty_io &&
               time_after(jiffies, bdi->wb.last_active +
                       bdi_longest_inactive())) {
                …
                //杀掉这个无用的回写 deamon
                action = KILL_THREAD;
                break;
```

```
            }
            spin_unlock(&bdi->wb_lock);
        }
        …
        //根据 action 执行需要的操作
        switch (action) {
        case FORK_THREAD:
            __set_current_state(TASK_RUNNING);
            /*为该 struct backing_dev_info 设备启动回写 deamon，参数为该设备的回
              写控制机构*/
            task = kthread_create(bdi_writeback_thread, &bdi->wb,
                        "flush-%s", dev_name(bdi->dev));
            …
            break;

        case KILL_THREAD:
            __set_current_state(TASK_RUNNING);
            //杀掉无用的 deamon
            kthread_stop(task);
            break;

        case NO_ACTION:
            //不需要回写，睡眠定时醒来
            if (!wb_has_dirty_io(me) || !dirty_writeback_interval)
                …
                schedule_timeout(bdi_longest_inactive());
            else
                schedule_timeout(msecs_to_jiffies(dirty_writeback_interval
                * 10));
            //检查自身是否需要 freeze
            try_to_freeze();
            /* Back to the main loop */
            continue;
        }
        …
    }

    return 0;
}
```

回写 deamon，bdi_writeback_thread 的作用与 bdi_forker_thread 类似，也是检查是否有脏页以执行回写。但是 bdi_writeback_thread 不同的地方是其仅针对当前 struct backing_dev_info 设备，而不是整个 struct backing_dev_info 链表。

8.4.3　回写机制的层次操作

回写的对象，不是针对某一个 page 或文件，回写针对的是一个设备。所以回写操作的层次是文件系统，然后是节点 struct address_space，最后才是 page。

```
//回写操作的实体，无论哪种回写时机都会使用到该函数
void writeback_inodes_wb(struct bdi_writeback *wb,
        struct writeback_control *wbc)
{
    int ret = 0;
…
    //只要该设备上有需要回写的 page
    while (!list_empty(&wb->b_io)) {
        …
        //针对该设备上文件系统操作，sb 即为超级块
        ret = writeback_sb_inodes(sb, wb, wbc, false);
        …
    }
    …
}

//针对一个文件系统内的回写
static int writeback_sb_inodes(struct super_block *sb, struct bdi_writeback
*wb,
        struct writeback_control *wbc, bool only_this_sb)
{
    /*wb 为当前设备的回写操作，其上的 b_io 队列是该设备上的文件系统的节点*/
    while (!list_empty(&wb->b_io)) {
        …
        //节点层次的回写
        writeback_single_inode(inode, wbc);
        …
    }
    …
}
```

8.4.4　节点层次的回写

Inode 是回写机制的基本单位，内核针对每个节点执行回写操作。节点回写包括两方面的动作：一是节点本身的回写，如在文件增长时节点的长度发生改变，这时须将节点本身回写；二是节点对应文件内容的回写，即脏页回写。这两个动作由一个函数框架来完成，代码如下：

```
/*
回写一个节点，包括节点本身和节点内容
 */
static int writeback_single_inode(struct inode *inode, struct writeback_
control *wbc)
{
    …
    //节点内容的回写，该函数将提交节点脏页
    ret = do_writepages(mapping, wbc);

    /*
     若控制结构指示为 WB_SYNC_ALL，在这里等待该节点 page 的 writeback 完成
     */
    if (wbc->sync_mode == WB_SYNC_ALL) {
        int err = filemap_fdatawait(mapping);
        …
    }
    …
    spin_lock(&inode->i_lock);
    /*
     I_DIRTY 指出 inode 本身被修改
     */
    dirty = inode->i_state & I_DIRTY;
    inode->i_state &= ~(I_DIRTY_SYNC | I_DIRTY_DATASYNC);
    spin_unlock(&inode->i_lock);
    /*这里将回写 inode 本身。作为文件系统元数据，只有具体的文件系统才知道，如何回写
      inode 本身，这里调用其超级块操作函数： int (*write_inode) (struct inode *,
      struct writeback_control *wbc); */
    if (dirty & (I_DIRTY_SYNC | I_DIRTY_DATASYNC)) {
        int err = write_inode(inode, wbc);
        …
    }
…
}

/*
基础的节点内容回写函数，主要执行具体文件系统提供的 writepages 函数，若具体的文件系统没
有提供该函数，则执行内核默认例程。事实上一些文件系统尽管提供了自己的 writepages 函数，
但其实现还是以调用内核默认例程为主。本文着重分析内核默认例程

*/
int do_writepages(struct address_space *mapping, struct writeback_control
*wbc)
{…
    //如果具体的文件系统提供 writepages 函数则使用
```

```
    if (mapping->a_ops->writepages)
        ret = mapping->a_ops->writepages(mapping, wbc);
    else
    //否则，使用内核默认的节点回写机制
        ret = generic_writepages(mapping, wbc);
    return ret;
}

/*将一个 inode 的页面提交到 block 层，mapping 为该 inode 对应的 page cache 树，wbc
  为当前写回操作的控制模式*/
int generic_writepages(struct address_space *mapping,
                struct writeback_control *wbc)
{
    struct blk_plug plug;
    int ret;

    /* 如果该 inode 所属文件系统没有实现 writepage 例程，则这种页面回写方式无法使用，
       不过对于大部分文件系统都基于内核基础函数 block_write_full_page 实现了该例程*/
    if (!mapping->a_ops->writepage)
        return 0;

    blk_start_plug(&plug);
/*将该 inode 所有状态为脏的页面收集起来，然后依次对其执行 lock 操作，然后调用 writepage
  例程将该页写回，在 writepage 例程里，在完成页面的 BH 提交后将对该 page unlock*/
    ret = write_cache_pages(mapping, wbc, __writepage, mapping);
    blk_finish_plug(&plug);
    return ret;
}


/*
将一个节点的脏页回写出去
*/
int write_cache_pages(struct address_space *mapping,
             struct writeback_control *wbc, writepage_t writepage,
             void *data)
{
    …

    //从该节点第一个脏页面开始回写脏页面，直到完成
    while (!done && (index <= end)) {
        …
        //在该节点的 page cache 里取出一定数量的脏页
        nr_pages = pagevec_lookup_tag(&pvec, mapping, &index, tag,
                min(end - index, (pgoff_t)PAGEVEC_SIZE-) + 1);
```

```
        …
        //依次处理取出的每一个 page
        for (i = 0; i < nr_pages; i++) {
            struct page *page = pvec.pages[i];
            …
            //检查该页是否为脏，否则也没有必要将其回写了
            if (!PageDirty(page)) {
                …
            }
            //该页已经处于写回状态，只要控制结构允许，等待其完成
            if (PageWriteback(page)) {
                if (wbc->sync_mode != WB_SYNC_NONE)
                    wait_on_page_writeback(page);
                else
                    goto continue_unlock;
            }

            //脏页监控机制的一环，参见脏页监控机制
            if (!clear_page_dirty_for_io(page))
                goto continue_unlock;
            …
            //调用具体文件系统 writepage，将脏页提交到 block 层
            ret = (*writepage)(page, wbc, data);
           …
            }
        …
        }
    …
}

/*将一个页面提交到 Block 层，get_block 为该文件系统的映射函数*/
int block_write_full_page(struct page *page, get_block_t *get_block,
            struct writeback_control *wbc)
{
    //实现主体是 static int __block_write_full_page(…)函数
    return block_write_full_page_endio(page, get_block, wbc,
}
```

将一个页提交给 Block 的说法不是很准确，向 Block 提交的基本单位是 BH，一个 page 对应可能对应多个 BH，但是重点分析基于 emmc+ext+4k 文件块的嵌入式及手机系统，所以这里一个 BH 对应一个 page。

```
/*
页面提交函数，该函数是内核为 writepage 提供的基础例程，供具体的文件系统选择使用
*/
```

```
static int __block_write_full_page(struct inode *inode, struct page *page,
            get_block_t *get_block, struct writeback_control *wbc,
            bh_end_io_t *handler)
{
    …
    //一个文件块的大小
    const unsigned blocksize = 1 << inode->i_blkbits;
    …
    //该文件最后一个文件逻辑块号
    last_block = (i_size_read(inode) -1) >> inode->i_blkbits;

    /*如果该 page 没有 BH，则为其创建，对于 ext4 文件系统，在这个工作在写框架的第一步
      就已经完成了*/
    if (!page_has_buffers(page)) {
        create_empty_buffers(page, blocksize,
                    (1 << BH_Dirty)|(1 << BH_Uptodate));
    }

    /* 当前 page 对应的起始逻辑文件块号*/

    block = (sector_t)page->index << (PAGE_CACHE_SHIFT - inode->i_blkbits);
    head = page_buffers(page);
    bh = head;

    /*针对每个文件逻辑块，寻找其对应的文件系统物理块号，对于不同文件系统这里有不同情
      形，而对于 ext4 系统，尽管在其写框架的第一步就做了映射，但是有的文件系统在页面
      提交时才会做映射，代码的这个位置需要提供较为 generic 操作，所以为了兼容别的文件
      系统。这里会再一次映射以取得物理块号，而对于 ext4 系统这里不过就是在 cache 里再
      取一次罢了，并不需要再次启动磁盘操作
     */
    do {
        if (block > last_block) {

            /* 已经越过了文件长度，但是还有新的 BH，这种情况发生在文件块大于或小于
               page，且文件总长度的最后一个文件块不能对应最后一个 page 的最后一个文件
               块。这是没有意义的文件块，清除其脏状态
             */
            clear_buffer_dirty(bh);
            set_buffer_uptodate(bh);
        } else if ((!buffer_mapped(bh) || buffer_delay(bh)) &&
               buffer_dirty(bh)) {
            //映射物理块，这一次一定要取得物理块号
            err = get_block(inode, block, bh, 1);
            if (err)
                goto recover;
```

```
            clear_buffer_delay(bh);
            if (buffer_new(bh)) {
                /*查看新分配的文件块是否在 bdev 文件系统节点里，metadata 占用物理块
                  的重新分配有可能导致这种情况*/
                clear_buffer_new(bh);
                unmap_underlying_metadata(bh->b_bdev,
                            bh->b_blocknr);
            }
        }
        bh = bh->b_this_page;
        block++;
    } while (bh != head);

    do {
        …

        if (wbc->sync_mode != WB_SYNC_NONE) {
        //控制结构要求真正写存储设备，lock 该 BH
            lock_buffer(bh);
        } else if (!trylock_buffer(bh)) {
        //控制结构没要求真正写存储设备，如果 lock 不住该 BH，将其重新置脏
            redirty_page_for_writepage(wbc, page);
            continue;
        }
        if (test_clear_buffer_dirty(bh)) {
        //将该 BH 的完成 handler 置为 end_buffer_async_write
            mark_buffer_async_write_endio(bh, handler);
        } else {
        //有人在 lock 该 BH 时已经把 BH 写进存储设备了
            unlock_buffer(bh);
        }
    } while ((bh = bh->b_this_page) != head);

    /*
     将要提交 Block 层，该 page 属于 writeback 状态
     */
    set_page_writeback(page);

    do {
        struct buffer_head *next = bh->b_this_page;
        if (buffer_async_write(bh)) {
            //提交到 Block 层
            submit_bh(write_op, bh);
            nr_underway++;
        }
```

```
        bh = next;
    } while (bh != head);
    //解锁该 page
    unlock_page(page);

    err = 0;
done:
    if (nr_underway == 0) {
        /*
        该 page 没有需要提交的 BH，这发生在有别的线程完成 BH 的写回动作，或者控制结构
        不要求 sync，且该页所有 BH 都无法 lock 的情况
         */
        end_page_writeback(page);
    }
    return err;
    …
}
```

在该页面的写回完成后，block 层会依次对完成 BH 调用 void end_buffer_async_write(…)函数，代码如下：

```
/*
 * Completion handler for block_write_full_page() - pages which are unlocked
 * during I/O, and which have PageWriteback cleared upon I/O completion.
 */
void end_buffer_async_write(struct buffer_head *bh, int uptodate)
{
    …
   //该 BH 对应的 page
    page = bh->b_page;
    if (uptodate) {
        /*更新 BH 状态为 uptodate，BH 的 uptodate 与 page 的 uptodate 不同，前者在写
          回完成才处于 update，后者在脏时就认为 update 了*/
        set_buffer_uptodate(bh);
    } else {
        //IO 出错，不考虑这种情况
        …
    }

    first = page_buffers(page);
    local_irq_save(flags);
    bit_spin_lock(BH_Uptodate_Lock, &first->b_state);
    //清除该 BH 的 async 标志，解锁该 BH
    clear_buffer_async_write(bh);
    unlock_buffer(bh);
    tmp = bh->b_this_page;
```

```
        //沿着该 page 的 BH 转一圈，看还有没有完成写回的 BH
        while (tmp != bh) {
            if (buffer_async_write(tmp)) {
                //还有 BH 在写回过程中
                goto still_busy;
            }
            tmp = tmp->b_this_page;
        }
        //该页所有的 BH 都完成了写回
        bit_spin_unlock(BH_Uptodate_Lock, &first->b_state);
        local_irq_restore(flags);
        //结束该 page 的 writeback 状态
        end_page_writeback(page);
        return;
    …
    }
```

以上分析的页提交是异步页提交，再提交到 block 层之后当前线程就去忙别的事情了，比如写回 deamon 和日志 deamon，它们不可能每次提交都等待完成之后再做下一步的工作，那样太耗时了。还有一种 block 提交是同步提交，即提交到 block 后该线程就等待该 BH 完成再做别的。

第 9 章　EXT4 文件系统

9.1　Android 文件系统的选择

在 Android 被大量采用嵌入式及手机系统中，文件系统的选择往往跟存储介质有直接关系。

若系统使用 nand 作为其存储介质。在 nand 上构建文件系统有两种方式，对于小规模容量的 nand 芯片（笔者认为在 512M 字节以下），可以选择 yffs2 作为其文件系统。对于较大容量的 nand 芯片，宜选用 UBIFS 作为其文件系统。这两种基于 nand 的文件系统之上都可以构建 Android 系统。

但是，Android 设备，无论如何都不能选择 Jffs2 作为其文件系统。其直接原因是 Jffs2 没有能够提供 Android 系统需要的 int (*fiemap)(s…);机制。其原因在于，不同于 yffs2 和 ubifs，Jffs2 直接架构在 mtd char 设备上，文件布局设计为每次写操作都作为一个新的数据节点附加在文件系统后部，工作时需要加载并维护所有的数据节点。每次写操作都涉及数据节点的分割和合并。且其实现的时候需要直接处理 nand 设备的特性，需要在文件系统层面实现磨损均衡、节点合并等操作，使得 Jffs2 维护其数据节点工作更加复杂。对于 VFS 读框架，Jffs2 还可以很好的适应。但是对于 VFS 写框架 Jffs2 就无法完整适配，每次写操作都需要直接操作到 Jffs2 维护的数据节点，以至于不能提供 writepage 机制，导致 filemap 依赖的脏页回写机制无法工作，自然无法支持 filemap 机制。尽管如此，作为最早被嵌入式 Linux 大量使用的 nand 文件系统，Jffs2 在小型、写操作不频繁的嵌入式系统中，由于其高效、稳定，仍不失最佳选择。

若系统使用 EMMC、SD 卡、SSD 等基于 nand 的块设备，由于这些设备的工作特性已经非常接近磁盘了，所以其上的文件系统的首选自然是 EXT4。作为传统的 EXT2/3 文件系统的演进，EXT4 是 Linux 社区支持最好的块设备文件系统之一，能全面发挥内核特性。

9.2　EXT4 文件节点

本节分析 EXT4 文件节点的基本结构、布局和获取方式。

9.2.1　EXT4 inode 基础结构

EXT4 文件系统的 inode 结构定义在 struct ext4_inode：

```
struct ext4_inode {
    __le16  i_mode;          /*File mode */
    __le16  i_uid;           /*Low 16 bits of Owner Uid */
    …
    __le16  i_gid;           /*Low 16 bits of Group Id */
   …
    /*普通文件或者目录文件用于文件块索引，特殊文件可以另作它用*/
    __le32  i_block[EXT4_N_BLOCKS];/*Pointers to blocks */
    …
};
如果用于文件块索引，i_block[]数组的使用分成 4 个部分：
//前 12 项直接映射
#define EXT4_NDIR_BLOCKS        12
//第 EXT4_IND_BLOCK 项指向间接映射
#define EXT4_IND_BLOCK          EXT4_NDIR_BLOCKS
//第 EXT4_DIND_BLOCK 项指向 2 次间接映射
#define EXT4_DIND_BLOCK         (EXT4_IND_BLOCK + 1)
//第 EXT4_TIND_BLOCK 项指向 3 次间接映射
#define EXT4_TIND_BLOCK         (EXT4_DIND_BLOCK + 1)
#define EXT4_N_BLOCKS           (EXT4_TIND_BLOCK + 1)
```

9.2.2 EXT4 raw inode 的定位

EXT4 raw inode 指的是 EXT4 文件节点在存储介上的表示，其布局可以通过其 inode 定位函数清楚的体现出来。int __ext4_get_inode_loc(…)是 EXT4 inode 的定位函数，该函数通过给定的 inode 号，将对应在 EXT4 文件系统里的 inode 提取出来。

```
//inode->i_ino 为该文件在 ext4 文件系统上的节点号
static int __ext4_get_inode_loc(struct inode *inode,
                strustruct ext4_iloc *iloc, int in_mem)
{
    struct ext4_group_desc  *gdp;
    struct buffer_head  *bh;
   //inode->i_sb;指向 ext4 文件系统的超级块
    struct super_block  *sb = inode->i_sb;
    ext4_fsblk_t        block;
    int         inodes_per_block, inode_offset;

    iloc->bh = NULL;

   //检查该节点是否为有效 ext4 节点号
    if (!ext4_valid_inum(sb, inode->i_ino))
        return -EIO;
```

```
    /*EXT4_INODES_PER_GROUP(sb)记录了一个块组里有多少节点，节点号除以块组节点数，
    得到该节点位于的块组号*/
    iloc->block_group = (inode->i_ino - 1) / EXT4_INODES_PER_GROUP(sb);

/*根据块组号取出块组描述符，该描述里记录了本块组的节点表的起始块号*/
    gdp = ext4_get_group_desc(sb, iloc->block_group, NULL);
    if (!gdp)
        return -EIO;

    /*
     inodes_per_block 表示每个 ext4  block 里能放多少节点
     */
    inodes_per_block = EXT4_SB(sb)->s_inodes_per_block;

   /*inode_offset 为节点号与每块组节点数的余数，即为节点号在自己所在块组的偏移量，
   等于把前面块组的节点数一一减去*/
    inode_offset = ((inode->i_ino - 1) %
            EXT4_INODES_PER_GROUP(sb));

    /*ext4_inode_table(sb, gdp)为该块组的 inode 节点表起始地址，inode_offset
    除以每块组所包含节点数可以得到自己所在块号 */
    block = ext4_inode_table(sb, gdp) + (inode_offset / inodes_per_block);

/*所在块号的偏移量，即为从所在块到自己节点之前的节点数*/
    iloc->offset = (inode_offset % inodes_per_block) * EXT4_INODE_SIZE(sb);

/*启动块设备驱动，把节点所在块取出来*/
    bh = sb_getblk(sb, block);

    …
has_buffer:
    iloc->bh = bh;
    return 0;
}
```

9.2.3　EXT4 inode 的获取

要处理文件就得知道文件在磁盘上的具体构造，这就需要获得其磁盘上的节点信息，为 VFS 提供节点获取函数是文件系统实现的必要工作。EXT4 提供的磁盘节点获取函数为 ext4_iget。工作时 VFS 通过 ext4_iget 函数来获取 EXT4 raw inode 的信息，并将其与 VFS 统一的节点管理机构 inode 接驳起来。

```
//根据超级块和节点号获取节点
struct inode *ext4_iget(struct super_block *sb, unsigned long ino)
```

```
{
    struct ext4_iloc iloc;
    //ext4 raw inode
    struct ext4_inode *raw_inode;
    struct ext4_inode_info *ei;
    //内核统一的文件节点管理——inode
    struct inode *inode;
    …
  /*导致 static struct inode *ext4_alloc_inode(struct super_block *sb)被调
  用，该函数分配 struct ext4_inode_info 结构，并返回其成员变量 struct inode
  vfs_inode;的指针。接着该 struct inode 被加入到 inode cache，而 inode cache
  是挂载超块上一个 hash 表*/
    inode = iget_locked(sb, ino);
    …
   //struct ext4_inode_info 被 ext4 节点分配函数分配
    ei = EXT4_I(inode);
    iloc.bh = NULL;

   //定位并读取 ino 所在的 block
    ret = __ext4_get_inode_loc(inode, &iloc, 0);
    if (ret < 0)
        goto bad_inode;
   //iloc->bh->b_data 为块起始地址，iloc->offset 为 raw inode 在 block 的偏移量
    raw_inode = ext4_raw_inode(&iloc);
   //把 raw inode 的信息放到 inode 里
    inode->i_mode = le16_to_cpu(raw_inode->i_mode);
    inode->i_uid = (uid_t)le16_to_cpu(raw_inode->i_uid_low);
    inode->i_gid = (gid_t)le16_to_cpu(raw_inode->i_gid_low);
    …
    //关键信息，文件块的 extent tree 的入口
    for (block = 0; block < EXT4_N_BLOCKS; block++)
        ei->i_data[block] = raw_inode->i_block[block];
    INIT_LIST_HEAD(&ei->i_orphan);
…
   /*ext4_file_operations、ext4_file_inode_operations、address_space_
   operations 三级文件操作函数指针，内核 MM 和 VFS 搭出了架子，具体实现由具体的文件系
   统来选择*/
    if (S_ISREG(inode->i_mode)) {
        //常规文件
        inode->i_op = &ext4_file_inode_operations;
        inode->i_fop = &ext4_file_operations;
        ext4_set_aops(inode);
    } else if (S_ISDIR(inode->i_mode)) {
        //目录文件
        inode->i_op = &ext4_dir_inode_operations;
```

```
            inode->i_fop = &ext4_dir_operations;
        } else if (S_ISLNK(inode->i_mode)) {
          //链接文件
            if (ext4_inode_is_fast_symlink(inode)) {
                inode->i_op = &ext4_fast_symlink_inode_operations;
                nd_terminate_link(ei->i_data, inode->i_size,
                    sizeof(ei->i_data) - 1);
            } else {
                inode->i_op = &ext4_symlink_inode_operations;
                ext4_set_aops(inode);
            }
        } else if (S_ISCHR(inode->i_mode) || S_ISBLK(inode->i_mode) ||
                S_ISFIFO(inode->i_mode) || S_ISSOCK(inode->i_mode)) {
            //设备文件，设备文件的 i_block[]数组有特殊用途
            inode->i_op = &ext4_special_inode_operations;
            if (raw_inode->i_block[0])
                init_special_inode(inode, inode->i_mode,
                  old_decode_dev(le32_to_cpu(raw_inode->i_block[0])));
            else
                init_special_inode(inode, inode->i_mode,
                  new_decode_dev(le32_to_cpu(raw_inode->i_block[1])));
        } else {
            …
        }

        /*用来读取 raw inode 的 struct buffer_head 使用完毕了，引用计数减一，参见块
        设备一章*/
        brelse(iloc.bh);
        ext4_set_inode_flags(inode);
        unlock_new_inode(inode);
        return inode;
        …
}
```

9.3　Mount

文件系统的加载已经被业界广泛研究，本书不赘述。对于 EXT4 文件系统，本文仅关心 EXT4 mount 过程的两个动作——文件超级块的获取和日志的加载。前者涉及 bdev 文件系统操作的一个实例，后者说明了文件系统结构。

```
static int ext4_fill_super(struct super_block *sb, void *data, int silent)
                __releases(kernel_lock)
                __acquires(kernel_lock)
```

```
{
…
    /*读出 EXT4 超级块，将整块分区作为一个 Inode 节点，基于 VFS  inode 操作的基础框架。
    不同的是不需要做文件逻辑块到磁盘物理块的映射，logical_sb_block 直接指出了超级块
    的位置*/
     if (!(bh = sb_bread(sb, logical_sb_block))) {
         …
     }
     /*
     bh->b_data 指向超级块的起始地址，offset 为超级块的偏移地址
      */
     es = (struct ext4_super_block *) (((char *)bh->b_data) + offset);
     …
     //EXT4 超级块里记录其文件块的大小
     blocksize = BLOCK_SIZE << le32_to_cpu(es->s_log_block_size);
     …
     //加载日志
     if (ext4_load_journal(sb, es, journal_devnum))
             goto failed_mount3;
…

}

/*根据 EXT4 文件系统的设计文档，其日志可以作为一个普通文件放在 EXT4 文件系统的内部，也
可以另辟一个块设备来存放*/

static int ext4_load_journal(struct super_block *sb,
                struct ext4_super_block *es,
                unsigned long journal_devnum)
{
     journal_t *journal;
    /*如果日志作为 EXT4 文件系统的一部分，s_journal_inum 记录了其文件号，否则其日志
    被放在块设备 s_journal_dev 中*/
     unsigned int journal_inum = le32_to_cpu(es->s_journal_inum);
     dev_t journal_dev;
     int err = 0;
     int really_read_only;
     …
     if (journal_inum) {
        //从 EXT4 文件系统内部将日志控制结构读取进来
         if (!(journal = ext4_get_journal(sb, journal_inum)))
             return-EINVAL;
     } else {
       /*从块设备 journal_dev 上将日志控制结构读取进来，操作 bdev 文件系统节点的又
       一实例*/
```

```
        if (!(journal = ext4_get_dev_journal(sb, journal_dev)))
            return -EINVAL;
    }
    …
    if (!err) {
        …
      //日志加载的主体，Recovery 工作在这里进行
        err = jbd2_journal_load(journal);
        …
    }
    …
    return 0;
}
```

9.4　EXT4 文件写操作

EXT4 写操作采用了内核提供的通用写例程 static size_t generic_perform_write(-)（不考虑 O_DIRECT 模式写），针对该例程 EXT4 需要为其提供如下子例程。

（1）ext4_da_write_begin：实现逻辑块到物理块的映射与分配、将文件对应内容读进内存，并启动日志操作。

```
static int ext4_da_write_begin(struct file *file, struct address_space
*mapping,
                    loff_t pos, unsigned len, unsigned flags,
                    struct page **pagep, void **fsdata)
{
    …
retry:
    //启动本次日志操作
    handle = ext4_journal_start(inode, 1);
    …
    //获取或创建对应的 page cache 树里的页面
    page = grab_cache_page_write_begin(mapping, index, flags);
    …
   //将该页面的内容读进内存
    ret = __block_write_begin(page, pos, len, ext4_da_get_block_prep);
    …
    return ret;
}
```

（2）ext4_da_write_end：向回写 deamon 提交脏页，并结束日志操作。

```
static int ext4_da_write_end(struct file *file,
                    struct address_space *mapping,
```

```
                   loff_t pos, unsigned len, unsigned copied,
                   struct page *page, void *fsdata)
{
    …
    //本次写对应的 handle
    handle_t *handle = ext4_journal_current_handle();

    …
    new_i_size = pos + copied;
    /*本次写操作导致了文件内容的增长，需要修改该节点 size 值，这就需要对元数据进行修改，
    而对于 EXT4 的 ordered 模式，数据是需要先于元数据被写入日志里的，所以在这里将 inode
    做特殊处理，使得日志 deamon 能够在合适的时机提交其数据页*/
    if (new_i_size > EXT4_I(inode)->i_disksize) {
        if (ext4_da_should_update_i_disksize(page, end)) {
            down_write(&EXT4_I(inode)->i_data_sem);
            if (new_i_size > EXT4_I(inode)->i_disksize) {
                //节点 size 改动的情况下检查是否是 order 模式
                if (ext4_should_order_data(inode))
              /*向日志系统提交该 inode，该 inode 被挂载进 t_inode_list 队列，这将
              导致该 jbd2 对于文件内容页的 block 层的写提交。由此可见，EXT4 ordered
              只在文件增长时才力图确保文件内容被写入文件系统的时间先于元数据被 JBD2
              日志的时间，这样才能保证恢复时不至于将别的文件内容误恢复进来，不仅文件
              系统结构错误而且将导致严重的安全漏洞。而在文件内容修改时，ordered 其
              元数据没有动，所以也没有必要将其挂在 t_inode_list 队列了*/
                    ret = ext4_jbd2_file_inode(handle,
                                inode);

                EXT4_I(inode)->i_disksize = new_i_size;
            }
            up_write(&EXT4_I(inode)->i_data_sem);
            …
        /*在日志里为该 inode 分配空间，并以 metadata 形式向日志提交该 inode，这将
        导致该 inode 节点的被写入日志*/
            ext4_mark_inode_dirty(handle, inode);
        }
    }
    /*将该页置为脏状态，若 inode size 发生改变，这里也将 inode 置为脏状态，这是正常的
    文件脏页提交，就相当于 JBD2 不存在一样*/
    ret2 = generic_write_end(file, mapping, pos, len, copied,
                        page, fsdata);
    …
    //结束并在需要的时候同步本次日志操作
    ret2 = ext4_journal_stop(handle);
    …
}
```

9.5　EXT4 journal

日志文件系统实现是一个各方面权衡的结果，既要保证文件系统吞吐速度又要保证其可恢复性。任何一种极端的追求都不是通用日志文件系统实现的目标。若要追求极端的可恢复性，需将导致大量冗余数据的写入，这将影响吞吐速度并且占用大量空间；同时导致文件碎片的大量的产生，这将使得每次文件系统的加载好用大量的内存和时间。笔者认为极端的可恢复性不是通用文件系统的应用领域，只有专门设计的软硬一体系统才能满足这个目标。而元数据在文件系统里起到提纲挈领的作用，元数据完好则系统极大程度上是可恢复的，对于常规嵌入式系统、手机系统，将文件系统元数据备份即可。而元数据使用的前提是其对应的文件数据能够安全，不然恢复出来的肯定是错误的数据；再者元数据所在位置一般不与数据连续在一起，将两者分开写无论对于磁盘上的文件系统物理布局还是 Block 层电梯算法都是相宜的，所以笔者认为 ordered 模式是最佳日志模式，本文仅讨论 ordered 模式，这也是大量被手机、嵌入式系统采用的模式。

JBD2 的日志操作的核心是 struct transaction_s 结构，它是文件系统与 JBD2 的交互的枢纽。一方面，文件系统将需要写入 JBD2 空间的元数据，相关联的文件数据挂在 struct transaction_s 结构相关链表上；另一方面，JBD2 的 kjournald2 线程把挂在 struct transaction_s 上的元数据列表，文件数据列表的操作列表取出来，或写入文件系统，或写入 JBD2 LOG 空间，并控制其写入顺序。

EXT4 的日志操作嵌在其写框架里，以一个文件内容写为例，EXT4 的 ordered 模式日志处理关键动作如下。

（1）在 write_begin(…)中调用 handle_t *ext4_journal_start(…)，创建当前 struct jbd2_journal_handle，并将其关联到当前活动的 struct transaction_s 结构。

（2）若发生文件增长，调用 int __ext4_handle_dirty_metadata(…)将牵连到元数据：包含块组的块使用位图，块组描述符的 BH 挂在 struct transaction_s 结构的 t_buffers 链表中。

（3）若发生文件增长，调用 static int ext4_ext_dirty(…)将牵连到元数据：包含 struct ext4_extent 或 struct ext4_extent_idx 的 BH 挂在 struct transaction_s 结构的 t_buffers 链表中。

（4）若发生文件增长，在 write_end(…)中将当前文件 inode 加入的当前活动 struct transaction_s 结构的 t_inode_list 列表。

（5）若发生文件增长，在 write_end(…)中调用 int ext4_mark_inode_dirty(…)，将当前文件 inode 本身当作元数据挂入 struct transaction_s 结构的 t_buffers 链表中。

（6）在 write_end(…)中调用 int jbd2_journal_stop(handle_t *handle)结束该 handle。若当前 handle 指出需要同步 handle->h_sync，则唤醒 kjournald2，并等待其同步完成。否则由 kjournald2 在合适的时机同步。

```
/*
 将需要修改 BH 的挂入 struct transaction_s 结构的相关链表中，jh 携带相关的 BH 指针，
 jlist 指出了需要挂载的目的链表，对于上文提到的元数据，list 值为 BJ_Metadata，对
 于这些元数据的 BH，EXT4 仅使用 bdev 文件系统的读，并未使用 bdev 文件系统的写，而将其直
```

```
 接提交给 JBD2
 */
void __jbd2_journal_file_buffer(struct journal_head *jh,
            transaction_t *transaction, int jlist)
{
    struct journal_head **list = NULL;
    int was_dirty = 0;
   //取出携带的 BH 指针
    struct buffer_head *bh = jh2bh(jh);

    …
  //根据 jlist 指出的类型将 BH 挂入对应的链表
    switch (jlist) {
    case BJ_None:
        J_ASSERT_JH(jh, !jh->b_committed_data);
        J_ASSERT_JH(jh, !jh->b_frozen_data);
        return;
    case BJ_Metadata:
      //元数据需要挂入 t_buffers 链表
        transaction->t_nr_buffers++;
        list = &transaction->t_buffers;
        break;
    case BJ_Forget:
        …
    case BJ_LogCtl:
        …
    case BJ_Reserved:
       //预留操作
        list = &transaction->t_reserved_list;
        break;
    }
   //挂入相关链表
    __blist_add_buffer(list, jh);
    jh->b_jlist = jlist;
…
}

/*
    将文件对应 inode 加入 struct transaction_s 结构的 t_inode_list 列表
 */
int jbd2_journal_file_inode(handle_t *handle, struct jbd2_inode *jinode)
{
   //从 handle 里找到当前 transaction
    transaction_t *transaction = handle->h_transaction;
    …
```

```
    jinode->i_transaction = transaction;
    //加入当前 transaction 的 t_inode_list 列表
    list_add(&jinode->i_list, &transaction->t_inode_list);
    …
    return 0;
}

//kjournald2 对文件内容页的处理
static int journal_submit_data_buffers(journal_t *journal,
        transaction_t *commit_transaction)
{
    struct jbd2_inode *jinode;
    int err, ret = 0;
    struct address_space *mapping;

    //保护 t_inode_list 链表
    spin_lock(&journal->j_list_lock);
    //依次取出自己 t_inode_list 链表上的节点
   list_for_each_entry(jinode, &commit_transaction->t_inode_list, i_list)
{
        //每一个节点都对应一个文件 page cache 树
        mapping = jinode->i_vfs_inode->i_mapping;
       //当前节点状态置为__JI_COMMIT_RUNNING，防止竞争线程的闯入
        set_bit(__JI_COMMIT_RUNNING, &jinode->i_flags);
        spin_unlock(&journal->j_list_lock);
        //写当前文件节点
        err = journal_submit_inode_data_buffers(mapping);
        if (!ret)
            ret = err;
        spin_lock(&journal->j_list_lock);
        //清楚该节点的__JI_COMMIT_RUNNING 状态
        clear_bit(__JI_COMMIT_RUNNING, &jinode->i_flags);
        smp_mb__after_clear_bit();
       //唤醒等待在该节点__JI_COMMIT_RUNNING 状态的线程
        wake_up_bit(&jinode->i_flags, __JI_COMMIT_RUNNING);
    }
    spin_unlock(&journal->j_list_lock);
    return ret;
}

/*节点 inode 的提交，该函数的实现使用内核提供的 inode 写的基本例程，是一个常规的文件内
容写*/
static int journal_submit_inode_data_buffers(struct address_space
*mapping)
{
```

```
    int ret;
   //写控制结构，要求做 SYNC
    struct writeback_control wbc = {
        .sync_mode =  WB_SYNC_ALL,
        .nr_to_write = mapping->nrpages * 2,
        .range_start = 0,
        .range_end = i_size_read(mapping->host),
    };
   //写该节点 page cache 树里的所有脏页
    ret = generic_writepages(mapping, &wbc);
    return ret;
}
```

在完成了文件内容页的写入之后，kjournald2 才会处理挂载在自己 t_buffers 上的元数据 BH，这样就保证了文件内容页先于元数据的写入。网上已经有了这方面的详细分析，请大家自己查阅，这里不再介绍。

9.6 Extent tree

Extent tree 是 EXT4 文件布局的基本结构，尽管 EXT4 也支持老式的间接块的方式，但是间接块存在自身占用过多磁盘空间、多大文件支持不好等缺点，所以 EXT4 的精髓之一在于其文件布局的改进，所以本书只讨论 Extent tree，不再讨论间接块的方式。

9.6.1 基础结构

本节描述 EXT4 文件系统的一些基础结构。

struct ext4_extent 记载一段连续的物理块到逻辑块的映射，代码如下：

```
struct ext4_extent {
    __le32  ee_block;       /*第一个逻辑块*/
    __le16  ee_len;         /*长度*/
    __le16  ee_start_hi;    /*高 16 位物理块地址 */
    __le32  ee_start_lo;    /*低 32 位物理块地址 */
};
```

这个结构说明了 ext4 的文件大小和文件系统的限制。

文件逻辑块地址跟物理块地址的关系类似于处理器的 32 位虚拟地址映射 40 位物理地址的逻辑关系。文件看到的是逻辑块，而在 ARM 32 位机器上，逻辑块最大是 4K，所以单个文件限制是 4G×4K=16T，物理块设备看到的 48 位索引地址，文件块的大小与逻辑块一致，所以文件系统限制为 2^{48}×4K。

struct ext4_extent_idx 指向 struct ext4_extent 表，代码如下：

```
struct ext4_extent_idx {
    __le32  ei_block;   /*对应 struct ext4_extent 表的其实逻辑块号 */
    __le32  ei_leaf_lo; /*struct ext4_extent 表的低 32 位*/
    __le16  ei_leaf_hi; /*struct ext4_extent 表的高 16 位*/
    __u16   ei_unused;
};
```

struct ext4_extent_header 位于 struct ext4_extent 表或者 struct ext4_extent_idx 表的表头，也位于节点 inode 结构。

```
struct ext4_extent_header {
    __le16  eh_magic;   /*probably will support different formats */
    /*struct ext4_extent 表或 struct ext4_extent_idx 表的实际项目数*/
    __le16  eh_entries;
    /*struct ext4_extent 表或 struct ext4_extent_idx 表的最大项目数*/
    __le16  eh_max;
    /*表示 struct ext4_extent 和 struct ext4_extent_idx 表的层叠数，如果位于节点
    inode 且为零，则表示节点 inode 中放的就是 struct ext4_extent*/
    __le16  eh_depth;
    __le32  eh_generation;
};
```

9.6.2　定位逻辑块的 struct ext4_extent

从逻辑块定位其 struct ext4_extent 的过程就是从 inode 节点穿过 struct ext4_extent_idx 表到达 struct ext4_extent 对应项的过程。

```
/*block 为需要映射的逻辑块号，path 为 struct ext4_ext_path 数组，用来记录从 inode
节点开始，需要寻找的 struct ext4_extent 和 struct ext4_extent_idx 对应项指针*/
struct ext4_ext_path * ext4_ext_find_extent(struct inode *inode, ext4_
lblk_t block,
                    struct ext4_ext_path *path)
{
    struct ext4_extent_header *eh;
    struct buffer_head *bh;
    short int depth, i, ppos = 0, alloc = 0;

/*即为(struct ext4_extent_header *) EXT4_I(inode)->i_data，在读取 EXT4 文件节
点的时候就把文件节点 i_block 位置复制到 i_data 里了，参见 ext4 inode 的获取*/
    eh = ext_inode_hdr(inode);
/*i_block 偏移 0x6 eh_depth，大于零表示 i_block 里是 struct ext4_extent_idx，等
于零表示 i_block 里是 struct ext4_extent*/
    depth = ext_depth(inode);

    /*account possible depth increase */
```

```
    …
   //初始 path 起始级
    path[0].p_hdr = eh;
    path[0].p_bh = NULL;

    i = depth;
    /*从 inode 开始逐级寻找 */
    while (i) {
        int need_to_validate = 0;
      …
      /*在当前 struct ext4_extent_idx 表中折半查找对应项，eh_depth 大于零的 inode
      也被认为是一种 struct ext4_extent_idx 表*/
        ext4_ext_binsearch_idx(inode, path + ppos, block);
        /*path[ppos].p_idx 记录下 struct ext4_extent_idx 项指向的下一级表物理块号*/
        path[ppos].p_block = ext4_idx_pblock(path[ppos].p_idx);
        path[ppos].p_depth = i;
        path[ppos].p_ext = NULL;
        //读取下一级索引表
        bh = sb_getblk(inode->i_sb, path[ppos].p_block);
        …
        //找到 struct ext4_extent_idx 表的表头
        eh = ext_block_hdr(bh);
        //级数递进，并记录表头级存储表的 BH
        ppos++;

        path[ppos].p_bh = bh;
        path[ppos].p_hdr = eh;
        i--;
        …
    }
  /*找到了 struct ext4_extent 表，其位置位于 path[ppos].p_bh */
    path[ppos].p_depth = i;
    path[ppos].p_ext = NULL;
    path[ppos].p_idx = NULL;

    /*在 struct ext4_extent 表折半查找对应项*/
    ext4_ext_binsearch(inode, path + ppos, block);
    /*if not an empty leaf */
    if (path[ppos].p_ext)
        path[ppos].p_block = ext4_ext_pblock(path[ppos].p_ext);

    ext4_ext_show_path(inode, path);

    return path;
```

```
...
}

/*struct ext4_extent 表的折半查找与 struct ext4_extent_idx 的查找方式类似*/

static void ext4_ext_binsearch(struct inode *inode,
        struct ext4_ext_path *path, ext4_lblk_t block)
{
   //定位表头
    struct ext4_extent_header *eh = path->p_hdr;
    struct ext4_extent *r, *l, *m;
   //检查实际表项
    if (eh->eh_entries == 0) {
        return;
    }

    //表起始项为 l
    l = EXT_FIRST_EXTENT(eh) + 1;
   //表起始项为 r
    r = EXT_LAST_EXTENT(eh);

    while (l <= r) {
      //m 为当前中间表项
        m = l + (r - l) / 2;
       //以中间表项的逻辑块号作为折半方向的依据
        if (block < le32_to_cpu(m->ee_block))
            r = m - 1;
        else
            l = m + 1;

    }
   /*返回最近的起始逻辑块号，小于当前待映射逻辑块的表项。由于其索引仅以 struct
   ext4_extent 起始逻辑块号作为依据，所以待映射逻辑块可能在返回的 struct
   ext4_extent 项以内，也能在其长度以外*/
    path->p_ext = l - 1;
    ...
}
```

9.6.3　定位逻辑块左右侧的 struct ext4_extent 项

当逻辑块不在 Extent tree 中，比如文件递增写时越过了块边界，或者写 Hole 时。这时需要定位该逻辑块左右的 struct ext4_extent 项。

```
/*定位左 struct ext4_extent 项，path 为上一节返回的定位路径*/
```

```
static int ext4_ext_search_left(struct inode *inode,
                struct ext4_ext_path *path,
                ext4_lblk_t *logical, ext4_fsblk_t *phys)
    {
    …
    depth = path->p_depth;
    *phys = 0;
    …
   /*定位左 struct ext4_extent 项比较简单，因为上一节返回 path 的最后一项就记录了起
   始逻辑块小于待映射逻辑块的 struct ext4_extent 项。若其长度无法覆盖，待映射的块那
   它就是左侧 struct ext4_extent 项*/
    ex = path[depth].p_ext;
    …

    *logical = le32_to_cpu(ex->ee_block) + ee_len - 1;
    *phys = ext4_ext_pblock(ex) + ee_len - 1;
    return 0;
}



/*定位右侧 struct ext4_extent 项，path 为上一节返回的定位路径。定位右侧 struct
ext4_extent 项有两种情况，若上节查找的起始逻辑块小于待映射逻辑块的 struct
ext4_extent 位于不是 struct ext4_extent 表的最后一样，那么其后面一项即为所找，但是
如果其为最后一项则需要逐级向上索引，再从上级 struct ext4_extent_idx 指向的 struct
ext4_extent 表取第一项*/
static int ext4_ext_search_right(struct inode *inode,
                 struct ext4_ext_path *path,
                 ext4_lblk_t *logical, ext4_fsblk_t *phys)
{
    …
    depth = path->p_depth;
    *phys = 0;

    if (depth == 0 && path->p_ext == NULL)
        return 0;

    //ex 为起始逻辑块小于待映射逻辑块的 struct ext4_extent 项
    ex = path[depth].p_ext;
    ee_len = ext4_ext_get_actual_len(ex);

    /*第一种情况很简答，直接取下一项即可*/
    if (ex != EXT_LAST_EXTENT(path[depth].p_hdr)) {
       //索引到下一项
         ex++;
```

```
        //右侧逻辑块
        *logical = le32_to_cpu(ex->ee_block);
//右侧物理块，记录下来，在块分配时有用
        *phys = ext4_ext_pblock(ex);
        return 0;
    }

    /*第二种情况，查询上一级，上一级若是最后一项，则说明需要的 struct ext4_extent
    项还要从再上一级索引*/
    while (--depth >= 0) {
        ix = path[depth].p_idx;
      //判断是否为最后一个 struct ext4_extent_idx
        if (ix != EXT_LAST_INDEX(path[depth].p_hdr))
            goto got_index;
    }

    /*找到最高级 struct ext4_extent_idx 项，向下找，逐级读取 struct ext4_extent_
    idx 表和定位 struct ext4_extent_idx 项*/
    return 0;

got_index:

    ix++;
   //取出最上级 struct ext4_extent_idx 里指向的物理块号
    block = ext4_idx_pblock(ix);
    //逐级递进
    while (++depth < path->p_depth) {
        //读出当前级的 struct ext4_extent_idx 表
        bh = sb_bread(inode->i_sb, block);
        if (bh == NULL)
            return -EIO;
        //定位表头
        eh = ext_block_hdr(bh);
        …
        //找第一项，往下寻找只需取第一项
        ix = EXT_FIRST_INDEX(eh);
        //下一级 struct ext4_extent_idx 表的位置
        block = ext4_idx_pblock(ix);
        …
    }
    //终于到了 struct ext4_extent 表，读取
    bh = sb_bread(inode->i_sb, block);
    …
    //定位表头
    eh = ext_block_hdr(bh);
```

```
        //取缔一项
        ex = EXT_FIRST_EXTENT(eh);
        //逻辑块号
        *logical = le32_to_cpu(ex->ee_block);
        //物理块号
        *phys = ext4_ext_pblock(ex);
        …
        return 0;
    }
```

9.7 块 分 配

块分配就是为一个新建的文件块找到磁盘上一个对应的存放位置。

9.7.1 块组的 buddy 算法

EXT4 尽量在同一个块组为一个 inode 分配物理块，每一个块组在存储设备上都有一个 bitmap 记录该块组的物理块使用情况，该 bitmap 里每一位对应一个物理块。为了更好地避免碎片，Linux 在实现 EXT4 的时候，使用了与其物理内存页级分配同样的伙伴算法。但是不同于页级物理内存的分配算法的实现，EXT4 通过层级的 bitmap 来实现对物理块的分割合并。

第一级 bitmap 即为块组的物理块 bitmap，每位记载一个物理块是否被使用。然后由此向上，产生 N 级 bitmap，每一级的 bitmap size 比上一级少一半，但是其 bitmap 里的每一位记载着 2^N 个连续物理块是否闲置，其中 2^{N-1} 为一个物理块大小，其原因在于一个块组正好用一个物理块存放其 bitmap。

除了第一级 bitmap 是直接从块组中直接读出来以外，其他级别的 bitmap 都是根据第一级 bitmap 生成的。

```
/*为某块组生成 buddy，group 为块组号*/
static noinline_for_stack int
ext4_mb_load_buddy(struct super_block *sb, ext4_group_t group,
                    struct ext4_buddy *e4b)
{
    int blocks_per_page;
    int block;
    int pnum;
    int poff;
    struct page *page;
    int ret;
    struct ext4_group_info *grp;
    struct ext4_sb_info *sbi = EXT4_SB(sb);
```

```
/*内核准备了一个特殊的文件来存放所有块组 buddy cache 的位图。该文件特殊之处在于内
核只利用到了其 page cache，其中的 page 不被 dirty，且当作 anon 页来处理，所以其仅
仅存在内存，关机即失*/
 struct inode *inode = sbi->s_buddy_cache;


 //计算每一个 page 对于多少个物理块
 blocks_per_page = PAGE_CACHE_SIZE / sb->s_blocksize;
 //索引块描述符
 grp = ext4_get_group_info(sb, group);

 …
 /*计算本块组的 bitmap 在 s_buddy_cache 的起始位置。每个块组需要两个物理块，一个放
 原始的 bitmap，一个放逐级生成的 bitmap，所以以物理块计算本块组第一级 bitmap 的起
 始位置为块组号 x2 */
 block = group * 2;
 /*每一页放多个块组，这里算出对应以页为单位起始位置*/
 pnum = block / blocks_per_page;
 /*如果块组较小，小于 2K，一页放置多于 2 个以上的块组 bitmap，这时需计算页内偏移量*/
 poff = block % blocks_per_page;

 /*常规的 page cache 索引操作*/
 page = find_get_page(inode->i_mapping, pnum);
 if (page == NULL || !PageUptodate(page)) {
        …
   /*第一次访问需要创建页面，这里页面分配属性为 GFP_NOFS，使得其页面与文件系统脱
   离关系*/
     page = find_or_create_page(inode->i_mapping, pnum, GFP_NOFS);
     if (page) {
         ….
         if (!PageUptodate(page)) {
             /*页面分配完毕要初始化该页面，这里是第一级 bitmap 需要从磁盘里读进
             来，参见下文分析*/
             ret = ext4_mb_init_cache(page, NULL);
             …
         }
         unlock_page(page);
     }
 }
 …
 }
 //bd_bitmap 为第一级 bitmap，计算其虚拟地址
 e4b->bd_bitmap_page = page;
 e4b->bd_bitmap = page_address(page) + (poff * sb->s_blocksize);
```

```
/*提升该页活跃度，对于无 SWAP 机制的手机、嵌入式系统，这里的操作没有实际意义，这种 anon
页是不会被 shrink 掉的*/
    mark_page_accessed(page);

   /*本块组其余 bitmap 在 s_buddy_cache 的定位*/
    block++;
    pnum = block / blocks_per_page;
    poff = block % blocks_per_page;

    //同样在 s_buddy_cache 节点的 page  cache 里索引
    page = find_get_page(inode->i_mapping, pnum);
    if (page == NULL || !PageUptodate(page)) {
         …
        //第一次索引，未遇，分配一个 GFP_NOFS page
        page = find_or_create_page(inode->i_mapping, pnum, GFP_NOFS);
        if (page) {
            …
            if (!PageUptodate(page)) {
            //初始化其余级别的 bitmap，参见下文
                ret = ext4_mb_init_cache(page, e4b->bd_bitmap);
                …
            }
            unlock_page(page);
        }
    }
    //记录其余级别的 bitmap 的地址
    e4b->bd_buddy_page = page;
    e4b->bd_buddy = page_address(page) + (poff * sb->s_blocksize);
    mark_page_accessed(page);
    …
    return 0;

}
```

```
/*初始化 buddy cache 的各级 bitmap，该函数被调用两次，第一次从磁盘读出块组的 bitmap，
第二次根据第一级 bitmap 生成其余级别的 bitmap*/

static int ext4_mb_init_cache(struct page *page, char *incore)
{
…

    inode = page->mapping->host;
    sb = inode->i_sb;
```

```
//该文件系统共有多少个块组
ngroups = ext4_get_groups_count(sb);
/*尽管是特殊的 inode，但是还是有 EXT4 superblock 节点分配函数生成，所以其
i_blkbits 与实际的节点相同*/
blocksize = 1 << inode->i_blkbits;
//每个页对应的物理块
blocks_per_page = PAGE_CACHE_SIZE / blocksize;

//一个块组要用到两个 block，所以除 2
groups_per_page = blocks_per_page >> 1;
if (groups_per_page == 0)
    groups_per_page = 1;

/*尽管小于 2K 的情况是存在的，但是大部分使用 EXT4 的手机、嵌入式系统的物理块都是 4K
了，所以一个 BH 即可 */
if (groups_per_page > 1) {
    //小于 2K 的情况
    i = sizeof(struct buffer_head *) * groups_per_page;
    bh = kzalloc(i, GFP_NOFS);
    …
} else
    bh = &bhs;

//一个 page 可能包含多个 block 的 bitmap，从第一个对齐
first_group = page->index * blocks_per_page / 2;

/*read all groups the page covers into the cache */
for (i = 0; i < groups_per_page; i++) {
    struct ext4_group_desc *desc;

    …
    //该块组描述符
    desc = ext4_get_group_desc(sb, first_group + i, NULL);
    …

    /*从块组描述符取出其 bitmap 所在的物理块号，然后在该文件系统所在块设备对应在
    bdev 文件系统里的节点中索引其 page cache 里的 page，然后取出对应的 BH，如果没
    有 page，则创建 page 再创建 BH，这里没有读*/
    bh[i] = sb_getblk(sb, ext4_block_bitmap(sb, desc));
    if (bh[i] == NULL)
        goto out;
    //该 Bitmap 已经 update 了下一个块组
    if (bitmap_uptodate(bh[i]))
        continue;
```

```
        lock_buffer(bh[i]);
        …
        ext4_lock_group(sb, first_group + i);
        //若该块组从来没有使用过
        if (desc->bg_flags & cpu_to_le16(EXT4_BG_BLOCK_UNINIT)) {
         //初始化该 bitmap，并写回磁盘
         ext4_init_block_bitmap(sb, bh[i],
                        first_group + i, desc);
         //认为 bitmap 和 bh 都 update 了
         set_bitmap_uptodate(bh[i]);
         set_buffer_uptodate(bh[i]);
         ext4_unlock_group(sb, first_group + i);
         unlock_buffer(bh[i]);
         continue;
        }
        ext4_unlock_group(sb, first_group + i);
        //如果读出的 BH 是 update 的
        if (buffer_uptodate(bh[i])) {
            /*更新该 bitmap 为 update */
            set_bitmap_uptodate(bh[i]);
            unlock_buffer(bh[i]);
            continue;
        }
        get_bh(bh[i]);
        /*到了这里说明该块组从来没有被读过，或者曾经被读进来但是又被 shrink 掉了，这
        时要提交读操作*/
        set_bitmap_uptodate(bh[i]);
        bh[i]->b_end_io = end_buffer_read_sync;
        submit_bh(READ, bh[i]);
    }

    /*对应刚才提交读操作的情况，这里要等待读操作完工*/
    for (i = 0; i < groups_per_page; i++)
        if (bh[i])
            wait_on_buffer(bh[i]);

    /*到了这里，块组 bitmap 的读操作已经完成，下面要生成 buddy cache 其余各级 bitmap*/
    first_block = page->index * blocks_per_page;
    for (i = 0; i < blocks_per_page; i++) {
        …
        //data 位于 buddy  cache 节点的 page  cache
        data = page_address(page) + (i * blocksize);
        //bitmap 是位于磁盘 bdev 节点的 page  cache
        bitmap = bh[group - first_group]->b_data;
        …
```

```
        //防止其余各级 bitmap 的块必为计数块
        if ((first_block + i) & 1) {
            …
            ext4_lock_group(sb, group);
            //首先将各级 bitmap 都置位
            memset(data, 0xff, blocksize);
            /*生成 buddy 其余各级 bitmap，见下文*/
            ext4_mb_generate_buddy(sb, data, incore, group);
            ext4_unlock_group(sb, group);
            incore = NULL;
        } else {
            //对于块组 bitmap 读，须将其内容复制到 buddy cache 节点中
            ext4_lock_group(sb, group);
            memcpy(data, bitmap, blocksize);

            /*扫描块组所有预分配块，并将预分配的长度记录在 bitmap 里，仅仅改动 budduy
            page cache，无块设备相关操作*/
            ext4_mb_generate_from_pa(sb, data, group);
            /*扫描块组所有未提交的自由块，并将预分配的长度记录在 bitmap 里，仅仅改动
            budduy page cache，无块设备相关操作*/
            ext4_mb_generate_from_freelist(sb, data, group);
            ext4_unlock_group(sb, group);
            …
            incore = data;
        }
    }
    SetPageUptodate(page);

…
}
/*Buddy cache 其余各级 bitmap 的生成操作，bitmap 为读进来的块组的 bitmap*/
static noinline_for_stack
void ext4_mb_generate_buddy(struct super_block *sb,
                void *buddy, void *bitmap, ext4_group_t group)
{
    struct ext4_group_info *grp = ext4_get_group_info(sb, group);
    ext4_grpblk_t max = EXT4_BLOCKS_PER_GROUP(sb);
    ext4_grpblk_t i = 0;
    ext4_grpblk_t first;
    ext4_grpblk_t len;
    unsigned free = 0;
    unsigned fragments = 0;
    unsigned long long period = get_cycles();

    /*在第一 bitmap 找出第一空闲块*/
```

```
    i = mb_find_next_zero_bit(bitmap, max, 0);
    grp->bb_first_free = i;
    while (i < max) {
        fragments++;
        //first 记录连续的第一个空闲块
        first = i;
        //i 记录连续的最后一个空闲块
        i = mb_find_next_bit(bitmap, max, i);
        //i 与 first 之间就是空闲块的长度
        len = i-first;
        free += len;
        if (len > 1)
            //这里开始构建 buddy cache 其余各级 bitmap
            ext4_mb_mark_free_simple(sb, buddy, first, len, grp);
        else
            grp->bb_counters[0]++;
        if (i < max)
            //继续找下一个空闲块
            i = mb_find_next_zero_bit(bitmap, max, i);
    }
    grp->bb_fragments = fragments;

    …

}


/*所谓伙伴算法就是把整个长度反复对折直到为 1，所以每一个空闲块的起点都必须是 2^N */
static void ext4_mb_mark_free_simple(struct super_block *sb,
                void *buddy, ext4_grpblk_t first, ext4_grpblk_t len,
                    struct ext4_group_info *grp)
{
    …

    border = 2 << sb->s_blocksize_bits;

    while (len > 0) {
        //从当前位置往后能满足 buddy 分割的长度
        max = ffs(first | border) - 1;
        /*计算整个长度为 2 的几次方 */
        min = fls(len) - 1;
       //当前能分割的长度与总长度之间取当前能分割的长度
      if (max < min)
            min = max;
        chunk = 1 << min;
```

```
        ...
       //当前位置对应的可分割长度，计算索引为（first >> min）
        if (min > 0)
            mb_clear_bit(first >> min,
                    buddy + sbi->s_mb_offsets[min]);

      //还剩下的长度
      len -= chunk;
      //
      first += chunk;
    }
}
```

以一个从块 2612 开始，长度为 30156 空闲为例，每次分割的结构如下：

当前起始位置	当前分割长度	当前级数
2612	4	2
2616	8	3
2624	64	6
2688	128	7
2816	256	8
3072	1024	10
4096	4096	12
8192	8192	13
16384	8192	13
24576	8192	13

```
//buddy  cache 其余各级 bitmap 的地址是在 ext4_mb_init 初始化的

int ext4_mb_init(struct super_block *sb, int needs_recovery)
{
...
    i = 1;
    offset = 0;
    max = sb->s_blocksize << 2;
    do {
        //从 i=1 开始，级数越高表示连续空闲块越长
        sbi->s_mb_offsets[i] = offset;
        sbi->s_mb_maxs[i] = max;
        //每一级的长度递减一倍
        offset += 1 << (sb->s_blocksize_bits - i);
        max = max >> 1;
        i++;
    } while (i <= sb->s_blocksize_bits + 1);
```

```
...
}


//查找对应 2^order 长度的位图
static void *mb_find_buddy(struct ext4_buddy *e4b, int order, int *max)
{
    char *bb;
    ...

    /*如果 order 为 0，则是寻找长度为 1 的连续块，使用的位图是对应块组 bitmap 的
    bd_bitmap */
    if (order == 0) {
        *max = 1 << (e4b->bd_blkbits + 3);
        return EXT4_MB_BITMAP(e4b);
    }
   /*否则以 sbi->s_mb_offsets 数组为基址,order 为索引去 buddy cache 里生成的 bitmap
   地址*/
    bb = EXT4_MB_BUDDY(e4b) + EXT4_SB(e4b->bd_sb)->s_mb_offsets[order];
    *max = EXT4_SB(e4b->bd_sb)->s_mb_maxs[order];

    return bb;
}

/*在对应的 bitmap 里寻找 block 位置处的连续块，这个函数的前提是已经在最低一级即对应块
组 bitmap 里探测 block 位置为空闲。这里 block 是物理块号的组内偏移量*/

static int mb_find_order_for_block(struct ext4_buddy *e4b, int block)
{
    int order = 1;
    void *bb;


    //直接使用 buddy cache 中其余各级 bitmap
    bb = EXT4_MB_BUDDY(e4b);
    while (order <= e4b->bd_blkbits + 1) {
      //每到一级 bitmap，block 都需除以 2，才能在下一级 bitmap 里索引到对应位置
        block = block >> 1;
        if (!mb_test_bit(block, bb)) {
            /*该位置置位表示对应的 2^order 连续块至少有一半被占用*/
            return order;
        }
        //该级位置为 0，表示对应的 2^order 连续块空闲
        bb += 1 << (e4b->bd_blkbits - order);
        //探测下一级
```

```
        order++;
    }
    return 0;
}
```

9.7.2　分配物理块

物理块分配的第一步是找出最理想的目标物理块。对于一个逻辑块，其最理想的物理块是连续紧接着自己上一个逻辑块对应的物理块，这样就能形成文件的连续布局，或者直接将在该文件 inode 节点所在块组位置作为目标物理块。

```
//找到逻辑块 block 最理想的物理块，path 为上文分析的 Extent tree 的索引结果
static ext4_fsblk_t ext4_ext_find_goal(struct inode *inode,
                struct ext4_ext_path *path,
                ext4_lblk_t block)
{
    …

    if (path) {
        struct ext4_extent *ex;
        depth = path->p_depth;

        /*直接取最终的 struct ext4_extent 项，而不是其左右 struct ext4_extent 项，
        因为从最终项才能找连续的物理块*/
        ex = path[depth].p_ext;
        if (ex) {
            //最终项的物理块起始地址
            ext4_fsblk_t ext_pblk = ext4_ext_pblock(ex);
            //最终项的逻辑块起始地址
            ext4_lblk_t ext_block = le32_to_cpu(ex->ee_block);

        /*如果该逻辑块大于最终项的逻辑块则取紧邻最终项的下一个物理块，这是最常见情况
        对应递增写文件，或者取紧邻最终项的前一个物理块*/
            if (block > ext_block)
                return ext_pblk + (block - ext_block);
            else
                return ext_pblk - (ext_block - block);
        }

        …
    }

    /*没有 path，直接去 inode 块组所在位置 */
    block_group = ei->i_block_group;
    …
```

```
    //取块组起始块位置
    bg_start = ext4_group_first_block_no(inode->i_sb, block_group);
    last_block = ext4_blocks_count(EXT4_SB(inode->i_sb)->s_es) - 1;

    //如果是 delay alloc，直接返回块起始地址
    if (test_opt(inode->i_sb, DELALLOC))
        return bg_start;

    /*否则理想块号被设置为块组起始物理块+逻辑块号+colour。其中逻辑块号为文件内逻辑块
    号，colour 为依据线程号散列值*/
    if (bg_start + EXT4_BLOCKS_PER_GROUP(inode->i_sb) <= last_block)
        colour = (current->pid % 16) *
            (EXT4_BLOCKS_PER_GROUP(inode->i_sb) / 16);
    else
        colour = (current->pid % 16) * ((last_block - bg_start) / 16);
    return bg_start + colour + block;
}

/*在确定块分配目标位置之后，要在目标位置 block 处寻找长度满足大于或等于 ex->fe_len 的
连续物理块，内核的使用函数 mb_find_extent 实现上述算法，该函数被内核调用时参数 order
都为 0*/

static int mb_find_extent(struct ext4_buddy *e4b, int order, int block,
                int needed, struct ext4_free_extent *ex)
{
    …
    //order==0，取出最低一级 bitmap，每位对应一个 block
    buddy = mb_find_buddy(e4b, order, &max);
    /*测试对应位置是否被占用，如果被占用，意味着 block 为目标的分配失败了*/
    if (mb_test_bit(block, buddy)) {
        ex->fe_len = 0;
        ex->fe_start = 0;
        ex->fe_group = 0;
        return 0;
    }

    /*block 位置处空闲，再往上寻找更大的连续空间*/
    if (likely(order == 0)) {
        order = mb_find_order_for_block(e4b, block);
      /*block 处对应的 2^order 连续空闲块，block 左移 order 位，才能在对应 bitmap
      里索引该连续块*/
        block = block >> order;
    }
    //记录下成果
    ex->fe_len = 1 << order;
```

```
    ex->fe_start = block << order;
    ex->fe_group = e4b->bd_group;

    /*若目标块与连续块起始地址有差值，只取从目标块开始的长度 */
    next = next - ex->fe_start;
    ex->fe_len -= next;
    ex->fe_start += next;

   //如果需要的长度不能够满足，跳到下一个 buddy 去找更多空间
    while (needed > ex->fe_len &&
          (buddy = mb_find_buddy(e4b, order, &max))) {

        if (block + 1 >= max)
            break;
        //跳到当前找到的连续空闲块的下一个 block
        next = (block + 1) * (1 << order);
        //从最低级的 bitmap 开始测试
        if (mb_test_bit(next, EXT4_MB_BITMAP(e4b)))
            break;
        //再次沿着下一个 buddy 的 block 一级一级往上找连续空闲块
        ord = mb_find_order_for_block(e4b, next);

        //又找到了一个连续空闲块，合并进来
        order = ord;
        block = next >> order;
        ex->fe_len += 1 << order;
    }
   //这是从 block 开始的连续空闲块
    return ex->fe_len;
}
```

ext4 基于目标块的物理块分配算法如下：

```
/*ac 为分配状态结构，里面记录了当前的块分配的状态，e4b 则是记录 buddy 使用的 bitmap
地址*/
static noinline_for_stack
int ext4_mb_find_by_goal(struct ext4_allocation_context *ac,
                struct ext4_buddy *e4b)
{
    ext4_group_t group = ac->ac_g_ex.fe_group;
    int max;
    int err;
    struct ext4_sb_info *sbi = EXT4_SB(ac->ac_sb);
    struct ext4_free_extent ex;

  //首先检查分配策略里是否允许基于目标表来分配
```

```
    if (!(ac->ac_flags & EXT4_MB_HINT_TRY_GOAL))
        return 0;

   //加载 buddy 使用的 bitmap，大多数情况下都可以命中 page cache
    err = ext4_mb_load_buddy(ac->ac_sb, group, e4b);
    if (err)
        return err;

   //锁住块组，在 ac 里已经记录了目标的块组
    ext4_lock_group(ac->ac_sb, group);
   /*acac->ac_g_ex 是记录目标块的分配信息， ac->ac_g_ex.fe_start 记录的是目标块
   号，ac->ac_g_ex.fe_len 记录了需要分配的长度*/
    max = mb_find_extent(e4b, 0, ac->ac_g_ex.fe_start,
                    ac->ac_g_ex.fe_len, &ex);
/*返回的 max 指出从 ac->ac_g_ex.fe_start 开始有多少个连续的物理块，下面要检查 max 是
否大于所需要的长度 ac->ac_g_ex.fe_len，如果满足，该 max 个 block 就被认为是最佳分配
结果，acac->ac_b_ex 记录着最佳分配结果*/

    if (max >= ac->ac_g_ex.fe_len && ac->ac_g_ex.fe_len == sbi->s_stripe)
{
        ext4_fsblk_t start;
     //ext4 新特性，stripe 分配
        start = ext4_group_first_block_no(ac->ac_sb, e4b->bd_group) +
            ex.fe_start;

        if (do_div(start, sbi->s_stripe) == 0) {
            ac->ac_found++;
            ac->ac_b_ex = ex;
            ext4_mb_use_best_found(ac, e4b);
        }
    } else if (max >= ac->ac_g_ex.fe_len) {
        //记录下最佳分配结果
        ac->ac_found++;
        ac->ac_b_ex = ex;
     /*这里有两个主要动作:(1)ac->ac_status 被置为新状态: AC_STATUS_FOUND ;(2)
     各级 buddy 的 bitmap 对应为被置位*/
        ext4_mb_use_best_found(ac, e4b);
    } else if (max > 0 && (ac->ac_flags & EXT4_MB_HINT_MERGE)) {
        /*尽管连续块不能满足长度，但 ac 分配策略允许合并，这种情况也认为 max 是最佳分
        配结果*/
        ac->ac_found++;
        ac->ac_b_ex = ex;
        ext4_mb_use_best_found(ac, e4b);
    }
    ext4_unlock_group(ac->ac_sb, group);
```

```
    ext4_mb_unload_buddy(e4b);

    return 0;
}
```

有了上文的铺垫，就可以分析 EXT4 最近基本块分配动作，代码如下：

```
/*这是 EXT4 常规的块分配函数，其策略是首先基于目标块分配算法在目标块组里分配所需的物理块。当这个尝试无法满足时再在别的块组里分配物理块*/
static noinline_for_stack int
ext4_mb_regular_allocator(struct ext4_allocation_context *ac)
{
    ext4_group_t ngroups, group, i;
    int cr;
    int err = 0;
    struct ext4_sb_info *sbi;
    struct super_block *sb;
    struct ext4_buddy e4b;

    sb = ac->ac_sb;
    sbi = EXT4_SB(sb);
    ngroups = ext4_get_groups_count(sb);
    …

    /*首先尝试目标块分配 */
    err = ext4_mb_find_by_goal(ac, &e4b);
   //大多数情况下，目标块分配都是有效的，直接返回即可
    if (err || ac->ac_status == AC_STATUS_FOUND)
        goto out;

    …
repeat:
    for (; cr < 4 && ac->ac_status == AC_STATUS_CONTINUE; cr++) {
        ac->ac_criteria = cr;
        //从目标块组开始
        group = ac->ac_g_ex.fe_group;
        //逐个块组寻找
        for (i = 0; i < ngroups; group++, i++) {
            …
            //针对每个块组都要加载 buddy cache 各级 bitmap
            err = ext4_mb_load_buddy(sb, group, &e4b);
            …
            //锁住该块组
            ext4_lock_group(sb, group);

            …
```

```
                //根据需要的长度使用不同的连续算法
                if (cr == 0)
                    //简单的直接扫描
                    ext4_mb_simple_scan_group(ac, &e4b);
                else if (cr == 1 && sbi->s_stripe &&
                        !(ac->ac_g_ex.fe_len % sbi->s_stripe))
              //基于 stripe 的分配
                    ext4_mb_scan_aligned(ac, &e4b);
                else
                    /*取该块组第一个物理块，从该物理块开始寻找最符合连续块，这与目标块分
                    配非常类似*/
                    ext4_mb_complex_scan_group(ac, &e4b);

                ext4_unlock_group(sb, group);
                ext4_mb_unload_buddy(&e4b);
             //找到需要的最佳连续物理块
                if (ac->ac_status != AC_STATUS_CONTINUE)
                    break;
            }
        }
        //使用最佳连续物理块
        if (ac->ac_b_ex.fe_len > 0 && ac->ac_status != AC_STATUS_FOUND &&
            !(ac->ac_flags & EXT4_MB_HINT_FIRST)) {

          //设置 buddy cache 的逐级 bitmap
            ext4_mb_try_best_found(ac, &e4b);
            …
        }
out:
        return err;
}
```

上文描述的是 EXT4 物理块分配的基本算法，而实际上 EXT4 物理块分配策略里还有个预分配机制。

首先每个 inode 有个 struct list_head i_prealloc_list;链表，里面存放着上文提到每次常规分配用不完的连续物理块，即最佳连续块与实际需要的物理块的差值。

其次系统把每次跨块组分配的用不完的连续物理块保存下来作为预分配的备用池，若在 inode 有个 struct list_head i_prealloc_list;链表没有合适物理块，则预分配算法会计算目标物理块与这些备用次里的块组距离，挑选最近的进行分配。

若预分配机制无法满足要求，才会启动常规物理块分配机制，且常规物理块分配完成后，又会把当次用不完物理块 cache 下来给预分配机制使用。

```
/*
```

```
 Ext4 物理块分配入口函数
 */
ext4_fsblk_t ext4_mb_new_blocks(handle_t *handle,
                struct ext4_allocation_request *ar, int *errp)
{
    int freed;
    struct ext4_allocation_context *ac = NULL;
    struct ext4_sb_info *sbi;
    …
    sb = ar->inode->i_sb;
    sbi = EXT4_SB(sb);

…
//分配 context，从头到尾记录分配状态的变化和各个状态的参数
    ac = kmem_cache_alloc(ext4_ac_cachep, GFP_NOFS);
    …
   /*通过 struct ext4_allocation_request 初始化 struct ext4_allocation_
   context，struct ext4_allocation_request 里记录了目标物理块号和长度*/
    *errp = ext4_mb_initialize_context(ac, ar);
    …

    //第一步启动预分配机制
    if (!ext4_mb_use_preallocated(ac)) {
       //预分配机制没有成功，转向常规分配机制
        ac->ac_op = EXT4_MB_HISTORY_ALLOC;
        ext4_mb_normalize_request(ac, ar);
repeat:
        /*启用常规分配机制*/
        *errp = ext4_mb_regular_allocator(ac);
        …
     //常规机制成功，且分到了多余的物理块
        if (ac->ac_status == AC_STATUS_FOUND &&
                ac->ac_o_ex.fe_len < ac->ac_b_ex.fe_len)
     //将多余的物理块 cache 在预分配机制里
            ext4_mb_new_preallocation(ac);
    }
    if (likely(ac->ac_status == AC_STATUS_FOUND)) {
        /*将块组的 bitmap 置位，这里与 buddy cache bitmap 无关，见下文分析*/
        *errp = ext4_mb_mark_diskspace_used(ac, handle, reserv_blks);
        …
    } else {
       //分配不成功，丢弃预分配块
        freed  = ext4_mb_discard_preallocations(sb, ac->ac_o_ex.fe_len);
        if (freed)
            goto repeat;
```

```
        *errp = -ENOSPC;
    }

    …
}
```

块组分配的最后一个动作是置位块组的块使用位图，这是操作 EXT4 文件系统元数据，要启动日志机制将元数据写入日志，代码如下：

```
//ac 记录了起始物理块、长度及块组相关
static noinline_for_stack int
ext4_mb_mark_diskspace_used(struct ext4_allocation_context *ac,
                handle_t *handle, unsigned int reserv_blks)
{
    struct buffer_head *bitmap_bh = NULL;
    struct ext4_group_desc *gdp;
    struct buffer_head *gdp_bh;
    struct ext4_sb_info *sbi;
    …

    sb = ac->ac_sb;
    sbi = EXT4_SB(sb);

    //读块组的块使用位图
    bitmap_bh = ext4_read_block_bitmap(sb, ac->ac_b_ex.fe_group);
    if (!bitmap_bh)
        goto out_err;


    err = ext4_journal_get_write_access(handle, bitmap_bh);
    …
    //取块组描述符，块组描述符所在 bh 为 gdp_bh
    gdp = ext4_get_group_desc(sb, ac->ac_b_ex.fe_group, &gdp_bh);
    …

    err = ext4_journal_get_write_access(handle, gdp_bh);
    …
    //ac->ac_b_ex 为块组内偏移量，这里计算实际物理块号
    block = ext4_grp_offs_to_block(sb, &ac->ac_b_ex);

    len = ac->ac_b_ex.fe_len;
    …
    //锁住块组
    ext4_lock_group(sb, ac->ac_b_ex.fe_group);
    //块组 bitmap 对应位置位，这导致 bitmap_bh 脏
    mb_set_bits(bitmap_bh->b_data, ac->ac_b_ex.fe_start,ac->ac_b_ex.fe
```

```
    _len);
    …
    len = ext4_free_blks_count(sb, gdp) - ac->ac_b_ex.fe_len;
   //修改该块组空闲物理块数目，这导致 gdp_bh 脏
    ext4_free_blks_set(sb, gdp, len);
    gdp->bg_checksum = ext4_group_desc_csum(sbi, ac->ac_b_ex.fe_group, gdp);

    ext4_unlock_group(sb, ac->ac_b_ex.fe_group);

    …
  /*bitmap_bh 与 gdp 都是 EXT4 的 metadata，且都被修改了，状态都处于脏（但是没置脏标
  志位及提交给回写机制）。EXT4 日志体系需要将 metadata 写入日志，所以以下要将 bitmap_bh
  与 gdp 交给日志系统处理。关于 jbd2 分析见相关章节*/
    err = ext4_handle_dirty_metadata(handle, NULL, bitmap_bh);
    if (err)
        goto out_err;
    err = ext4_handle_dirty_metadata(handle, NULL, gdp_bh);

    …
}
```

9.8　逻辑块到物理块的映射

EXT4 可以使用传统的多级间接块或者使用 Extent tree 机制来组织文件块，前者可以方便地兼容 EXT2/3 文件系统，但是效率太低，且元数据占用大量空间，所以目前即使在移动、嵌入式系统 EXT4 都使用 Extent tree 来组织其文件块。

逻辑块到物理块的定位有以下两种情况。

（1）已分配物理块的逻辑块的定位。

（2）逻辑块尚未分配物理块，需为其分配物理块且构建 Extent tree 各级结构。

```
/*
基于 Extent tree 的逻辑块映射物理块
 */


int ext4_ext_map_blocks(handle_t *handle, struct inode *inode,
            struct ext4_map_blocks *map, int flags)
{
    struct ext4_ext_path *path = NULL;
    struct ext4_extent newex, *ex;
    …
    struct ext4_allocation_request ar;
    ext4_io_end_t *io = EXT4_I(inode)->cur_aio_dio;
```

```
struct ext4_map_blocks punch_map;


/*ext4 有一个简单但是非常有效的 struct ext4_extcache 机制，每个 inode 挂载了
一个 struct ext4_ext_cache 类型的 i_cached_extent 变量，用来记录最近使用的
struct ext4_ext，每次进入 Extent tree 之间都检查逻辑块是否落入这个区间*/
if (ext4_ext_in_cache(inode, map->m_lblk, &newex) &&
    ((flags & EXT4_GET_BLOCKS_PUNCH_OUT_EXT) == 0)) {
    …
        /*文件逻辑块号减去 struct ext4_ext，起始逻辑块号再加上 struct
        ext4_ext 物理块号即得对应物理块号 */
        newblock = map->m_lblk
                - le32_to_cpu(newex.ee_block)
                + ext4_ext_pblock(&newex);
        /*从 map->m_lblk 往后的物理块长度 */
        allocated = ext4_ext_get_actual_len(&newex) -
            (map->m_lblk - le32_to_cpu(newex.ee_block));
        goto out;
    …
}

/*没有落入 i_cached_extent 区间，进入 Extent tree，参见上文分析 */
path = ext4_ext_find_extent(inode, map->m_lblk, NULL);
…
//取出 extent tree 的深度
depth = ext_depth(inode);

//path 的最后一项是对应的 struct ext4_ext

ex = path[depth].p_ext;
if (ex) {
    //struct ext4_ext 起始逻辑块
    ext4_lblk_t ee_block = le32_to_cpu(ex->ee_block);
    //struct ext4_ext 起始物理块号
    ext4_fsblk_t ee_start = ext4_ext_pblock(ex);
    unsigned short ee_len;
    //struct ext4_ext 记载的长度
    ee_len = ext4_ext_get_actual_len(ex);
    /*检查 map->m_lblk 是否落入这个区间，对于文件指针小于或等于文件长度的读操作
    该条件都成立 */
    if (in_range(map->m_lblk, ee_block, ee_len)) {
      //计算出物理块号
        newblock = map->m_lblk - ee_block + ee_start;
        /*map->m_lblk 以后的长度 */
        allocated = ee_len - (map->m_lblk - ee_block);
```

```
            ...

            }
        ...
    }

    /*
    到了这里说明是写操作越过了文件结尾处
     */
    if ((flags & EXT4_GET_BLOCKS_CREATE) == 0) {
        /*
         如果不能创建新的 block，只有返回了
         */
        ext4_ext_put_gap_in_cache(inode, path, map->m_lblk);
        goto out2;
    }
    /*
     物理块分配
     */

    /*找出左右 extents 紧邻的物理块，参见 Extree tree 一节*/
    ar.lleft = map->m_lblk;
    err = ext4_ext_search_left(inode, path, &ar.lleft, &ar.pleft);
    if (err)
        goto out2;
    ar.lright = map->m_lblk;
    err = ext4_ext_search_right(inode, path, &ar.lright, &ar.pright);
    if (err)
        goto out2;

    /*
    若 map->m_len 大于 EXT_INIT_MAX_LEN 或 EXT_UNINIT_MAX_LEN，需要将其与
    EXT_INIT_MAX_LEN 或 EXT_UNINIT_MAX_LEN 对齐
     */
    if (map->m_len > EXT_INIT_MAX_LEN &&
        !(flags & EXT4_GET_BLOCKS_UNINIT_EXT))
        map->m_len = EXT_INIT_MAX_LEN;
    else if (map->m_len > EXT_UNINIT_MAX_LEN &&
         (flags & EXT4_GET_BLOCKS_UNINIT_EXT))
        map->m_len = EXT_UNINIT_MAX_LEN;

    /*初始化一个新的 extent */
    newex.ee_block = cpu_to_le32(map->m_lblk);
    newex.ee_len = cpu_to_le16(map->m_len);
  //检查该 extent 是否与其他 extent 项重叠
```

```
    err = ext4_ext_check_overlap(inode, &newex, path);
    if (err)
        allocated = ext4_ext_get_actual_len(&newex);
    else
        allocated = map->m_len;

    /*初始化 struct ext4_allocation_request 结构 */
    ar.inode = inode;
  /*找到最相邻的物理块*/
    ar.goal = ext4_ext_find_goal(inode, path, map->m_lblk);
    ar.logical = map->m_lblk;
    ar.len = allocated;
    …
    //EXT4 物理块分配，参见上一节
    newblock = ext4_mb_new_blocks(handle, &ar, &err);
    …
   //物理块分配完毕，将物理块记录在 struct ext4_extent
    ext4_ext_store_pblock(&newex, newblock);
   //记录所需长度
    newex.ee_len = cpu_to_le16(ar.len);
    …
   /*将 struct ext4_extent 插入到 Extent tree，该操作会首先检查该 struct
   ext4_extent 是否可以和邻近的 struct ext4_extent 合并，如果可以合并，直接修
   改紧邻的 struct ext4_extent 长度即可。若不能合并，则创建一个新的 struct
   ext4_extent 并将其挂入 Extent tree。无论如何都是修改了 metadata，将导致
  ext4_ext_dirty 的调用，从而触发日志操作*/
    err = ext4_ext_insert_extent(handle, inode, path,
                        &newex, flags);
    …
}
```

第 10 章　RCU

Spin_lock 就好比单行道口的红灯，处理器遇到就得刹车，随着处理器核心越来越多，刹车的时机呈指数增长的态势。为了使得系统顺畅运行，要尽可能地取消单行道，RCU 就好比立交桥，处理器按照自己的道路顺畅行驶，不必打扰别人。

10.1　RCU Tree

10.1.1　RCU Tree 结构

为实现 RCU 结构的组织，内核提供了 Rcutiny 和 Rcutree 两种结构，前者适用于处理器个数较少的情况，Rcutiny 简单精悍易于分析，本书不做详细展开，Rcutree 用于大型系统，可管理数百颗处理器。Rcutree 树的叶子节点由 struct rcu_data 表示，该结构代表一个处理器，中间节点由 struct rcu_node 表示，根节点由 struct rcu_state 表示。

RCU 数层级的逻辑结构如下：

（1）如果 CPU 总数小于 struct rcu_node 的扇出系数，一层就足以容纳下所有的 CPU。

（2）扇出系数<CPU 总数≤扇出系数平方，两层的 tree 才能容纳下所有的 CPU。

（3）扇出系数立方<CPU 总数≤扇出系数 4 次平方，三层的 tree 才能容纳下所有的 CPU。

（4）扇出系数 4 次平方<CPU 总数，四层的 tree 才能满足。

这个层级结构在 struct rcu_state 定义中得到体现，代码如下：

```
struct rcu_state {
    //这个数组存放所有的 struct rcu_node
    struct rcu_node node[NUM_RCU_NODES];    /*Hierarchy*/
    //这个数组的每一项指向对应层的第一个 struct rcu_node
    struct rcu_node *level[NUM_RCU_LVLS];   /*Hierarchy levels*/
    //记录了每一层里有几个 struct rcu_node
    u32 levelcnt[MAX_RCU_LVLS + 1];         /*# nodes in each level*/
    u8 levelspread[NUM_RCU_LVLS];           /*kids/node in each level*/
    …
}
```

每颗 CPU 对应的管理结构 struct rcu_data 中定义了 4 个重要的 RCU 队列。

```
struct rcu_data{…
struct rcu_head **nxttail[RCU_NEXT_SIZE];
…}
```

其中定义了以下 4 个 RCU 队列。

```
#define RCU_DONE_TAIL       0   /*Also RCU_WAIT head. */
#define RCU_WAIT_TAIL       1   /*Also RCU_NEXT_READY head. */
#define RCU_NEXT_READY_TAIL 2   /*Also RCU_NEXT head. */
#define RCU_NEXT_TAIL       3
```

其中：RCU_DONE_TAIL 是已经过了 grace time 的 RCU 函数队列。

RCU_WAIT_TAIL 是等待当前的 grace time 的函数队列。

RCU_NEXT_READY_TAIL 是下一批需要等待 grace time 的函数队列，因为 RCU_NEXT_TAIL 是随时会增加的，所以增加这个队列。

RCU_NEXT_TAIL 是每次新加的函数被放入这个队列。

10.1.2 RCU Tree 的构建

RCU Tree 的初始化是根据处理器核心数，将上一节的层级结构实体化。这是在系统启动之初就需要完成的工作。

```
static void __init rcu_init_one(struct rcu_state *rsp)
{
    ...
    /*首先把 struct rcu_node *level[]中每个元素指向对应的每一层的第一个 struct
    rcu_node*/
    for (i = 1; i < NUM_RCU_LVLS; i++)
        rsp->level[i] = rsp->level[i - 1] + rsp->levelcnt[i - 1];

    //该函数计算出每一层的 struct rcu_node 对应多少颗 CPU
    rcu_init_levelspread(rsp);

    //下面从底往上，初始化整个 RCU tree 的每个 struct rcu_node 结点

    for (i = NUM_RCU_LVLS - 1; i >= 0; i--) {
        //首先找到该层每个 struct rcu_node 下对应多少颗 CPU
        cpustride *= rsp->levelspread[i];
        //找到这一层的第一个 struct rcu_node
        rnp = rsp->level[i];
        for (j = 0; j < rsp->levelcnt[i]; j++, rnp++) {
            raw_spin_lock_init(&rnp->lock);
            ...
            rnp->gpnum = 0;
            rnp->qsmask = 0;
            rnp->qsmaskinit = 0;
             /*该 struct rcu_node 对应索引值最小的 CPU，每一层的 cpustride 值都
             不同*/
            rnp->grplo = j * cpustride;
```

```
            //该 struct rcu_node 对应索引值最大的 CPU
            rnp->grphi = (j + 1) * cpustride - 1;
            if (rnp->grphi >= NR_CPUS)
                rnp->grphi = NR_CPUS - 1;
            if (i == 0) {
                //到了顶层
                rnp->grpnum = 0;
                …
            } else {
            /*该 struct rcu_node 在上一层 struct rcu_node 结点中索引，j 实际上是
            该层的索引，rsp->levelspread[i - 1]是上一层 struct rcu_node 结点管
            理多少个本层结点，通过取余即可*/
                rnp->grpnum = j % rsp->levelspread[i - 1];
            //对应的 mask 位
                rnp->grpmask = 1UL << rnp->grpnum;
                //记录下父节点是什么
                rnp->parent = rsp->level[i - 1] +
                         j / rsp->levelspread[i - 1];
            }
            //记录下该 struct rcu_node 所在层数
            rnp->level = i;
            INIT_LIST_HEAD(&rnp->blocked_tasks[0]);
            …
        }
    }
}

//接驳每个 CPU 的 rcu_sched_data 到对应的 struct rcu_node

#define RCU_INIT_FLAVOR(rsp, rcu_data) \
do { \
    …
   //初始化 struct rcu_state
    rcu_init_one(rsp); \
   //最底层的第一个结点
    rnp = (rsp)->level[NUM_RCU_LVLS - 1]; \
    j = 0; \
    for_each_possible_cpu(i) { \
    //i 是 CPU 的索引，j 是 struct rcu_node 结点的索引
        if (i > rnp[j].grphi) \
            j++; \
        //将每个 CPU 的 rcu_sched_data 的 mynode 指向对应的结点
        per_cpu(rcu_data, i).mynode = &rnp[j]; \
        //在 struct rcu_state 里记录下该 CPU
        (rsp)->rda[i] = &per_cpu(rcu_data, i); \
```

```
            rcu_boot_init_percpu_data(i, rsp); \
        } \
    } while (0)


    //进一步初始化每个 CPU 的 rcu_sched_data 结构
    static void __init
    rcu_boot_init_percpu_data(int cpu, struct rcu_state *rsp)
    {
        unsigned long flags;
        int i;
        //找到处理器对应的数据
        struct rcu_data *rdp = rsp->rda[cpu];
        //取出 struct rcu_node 树的根结点
        struct rcu_node *rnp = rcu_get_root(rsp);

        raw_spin_lock_irqsave(&rnp->lock, flags);
        //算出该 CPU 在其对应 struct rcu_node 里的 mask
        rdp->grpmask = 1UL << (cpu - rdp->mynode->grplo);
        rdp->nxtlist = NULL;
        //初始化该 CPU 的 RCU 队列
        for (i = 0; i < RCU_NEXT_SIZE; i++)
            rdp->nxttail[i] = &rdp->nxtlist;
        …
        raw_spin_unlock_irqrestore(&rnp->lock, flags);
    }
```

以上只是静态初始 CPU 及 struct rcu_node 树的静态关系。事实上这时候该 CPU 还没有被激活，在该 CPU 被激活后调用函数 static void __cpuinit rcu_online_cpu(int cpu)，进一步完成初始化。

10.2 Grace Period

10.2.1 Grace Period 的检测

每个处理器都经过至少一次 quiescent state，称为 grace period。处理器处于 quiescent state 时机如下：

（1）进入调度器。

（2）用户态，线程处在用户态。

（3）处理器运行 idle 线程或进入没有 tick 的状态 idle。

每颗处理器在自己的 RCU_SOFTIRQ 中，若自己经过了一次 quiescent state，则需要向系统报告，代码如下：

```
static void
rcu_check_quiescent_state(struct rcu_state *rsp, struct rcu_data *rdp)
{
    /*在当前处理器等待 quiescent state 或 RCU_SOFTIRQ 之前，系统启动了新一轮 grace
    period 的等待。该函数将当前处理器对应结构 struct rcu_data 的成员变量 gpnum 更新
    为新一轮的 grace period 号，且将 qs_pending 置 1*/
    if (check_for_new_grace_period(rsp, rdp))
        return;

    /*新一轮 grace period 被启动，qs_pending 被置 1   */
    if (!rdp->qs_pending)
        return;

    /*
     在经过 quiescent state 时该值被置 1
     */
    if (!rdp->passed_quiesc)
        return;

    /*
     向上一级即 struct rcu_node 结构汇报，当前处理器已经经过一个 quiescent state
     状态
     */
    rcu_report_qs_rdp(rdp->cpu, rsp, rdp, rdp->passed_quiesc_completed);
}


/*最底层 struct rcu_node 结构要检查是否自己属下所有处理器都经过 quiescent state 状
态，如果该条件成立，则逐级向上层 struct rcu_node 结构汇报*/
static void
rcu_report_qs_rdp(int cpu, struct rcu_state *rsp, struct rcu_data *rdp, long
lastcomp)
{
    unsigned long flags;
    unsigned long mask;
    struct rcu_node *rnp;
   //当前处理器直属一级的 struct rcu_node 结构

    rnp = rdp->mynode;
    raw_spin_lock_irqsave(&rnp->lock, flags);
   /*检查当前处理器经过 quiescent state 号是否是该 struct rcu_node 结构正在检测的
*/
    if (lastcomp != rnp->completed) {

        /*
```

```
          这种情况发生在刚刚有别的处理器抢先启动了新一轮 grace period 的等待。这时当
          前处理经过 quiescent state 当作无效，重新等待新一轮 grace period
          */
         rdp->passed_quiesc = 0; /*try again later! */
         raw_spin_unlock_irqrestore(&rnp->lock, flags);
         return;
     }
    /*grpmask 记录了当前处理器在其直属 struct rcu_node 结构里的对应位*/
     mask = rdp->grpmask;
     if ((rnp->qsmask & mask) == 0) {
         raw_spin_unlock_irqrestore(&rnp->lock, flags);
     } else {
   /*当前处理器在直属 struct rcu_node 结构，认可当前处理经过 quiescent state 有效*/
         rdp->qs_pending = 0;

         /*
          当前处理器挂载的 RCU 递进，但并不意味着这些 RCU 可以执行了，因为全部处理器状
          态还没有全部检查完
          */
         rdp->nxttail[RCU_NEXT_READY_TAIL] = rdp->nxttail[RCU_NEXT_TAIL];
         //向中间级 struct rcu_node 结构报告
         rcu_report_qs_rnp(mask, rsp, rnp, flags); /*rlses rnp->lock */
     }
}

/*中间层向 struct rcu_node 结构的 quiescent state 状态检测，若当前 struct rcu_node
结构所管辖的所有处理器的 quiescent state 状态有效意味着可以向上一级 struct rcu_node
结构汇报*/

static void rcu_report_qs_rnp(unsigned long mask, struct rcu_state *rsp,
          struct rcu_node *rnp, unsigned long flags)
    __releases(rnp->lock)
{
     struct rcu_node *rnp_c;

     /*逐级向 struct rcu_node 结构汇报 */
     for (;;) {
       …
       /*清除当前层对应的上一层 struct rcu_node 对应位*/
         rnp->qsmask &= ~mask;
         if (rnp->qsmask != 0 || rcu_preempt_blocked_readers_cgp(rnp)) {
        /*当前层对应的上一层 struct rcu_node 对应位已经被清除，这种情况发生在同级，
        struct rcu_node 所管辖的处理器中还有未达到 quiescent state 状态*/
             raw_spin_unlock_irqrestore(&rnp->lock, flags);
             return;
```

```
        }
        mask = rnp->grpmask;
        …

    /*同级 struct rcu_node 所管辖的处理器中均达到 quiescent state 状态，需向上
    层报告*/
        rnp_c = rnp;
        //当前 struct rcu_node 结构的父节点
        rnp = rnp->parent;
        raw_spin_lock_irqsave(&rnp->lock, flags);
        …
    }

    /*
    顶层 struct rcu_node 所管辖处理器均达到 quiescent state 状态，需向根节点 struct
    rcu_state 报告
     */
    rcu_report_qs_rsp(rsp, flags); /*releases rnp->lock. */
}

//struct rcu_state 控制着全局 grace period 的更新
static void rcu_report_qs_rsp(struct rcu_state *rsp, unsigned long flags)
    __releases(rcu_get_root(rsp)->lock)
{
    …
   //更新完成 GP 号
    rsp->completed = rsp->gpnum;
    rsp->signaled = RCU_GP_IDLE;
   //重新启动一个新的 GP 号
    rcu_start_gp(rsp, flags);  /*releases root node's rnp->lock*/
}
```

10.2.2　重新启动新一轮 Grace Period

启动新一轮 Grace Period 的关键是要将整个 RCU tree 重置，这样每个处理器到达后才能顺利向上汇报。

```
/*启动新一轮的 GP*/
static void
rcu_start_gp(struct rcu_state *rsp, unsigned long flags)
    __releases(rcu_get_root(rsp)->lock)
{
    …

    /*GP 号更新 */
```

```
    rsp->gpnum++;
    rsp->signaled = RCU_GP_INIT; /*Hold off force_quiescent_state. */
    rsp->jiffies_force_qs = jiffies + RCU_JIFFIES_TILL_FORCE_QS;
   //记录启动时间
    record_gp_stall_check_time(rsp);

    /*对于只有一层 struct rcu_node 结构的情况*/
    if (NUM_RCU_NODES == 1) {
        rcu_preempt_check_blocked_tasks(rnp);
        //把唯一个 struct rcu_node 更新至初始状态即可
        rnp->qsmask = rnp->qsmaskinit;
        //GP 号更新
        rnp->gpnum = rsp->gpnum;
        //完成 GP 号更新
        rnp->completed = rsp->completed;
        rsp->signaled = RCU_SIGNAL_INIT; /*force_quiescent_state OK*/
        /*GP 号更新且当前处理器对应 struct rcu_node 结构的 completed 被更新，更新
        处理器的 nxttail 数组指针*/
        rcu_start_gp_per_cpu(rsp, rnp, rdp);
        …
        raw_spin_unlock_irqrestore(&rnp->lock, flags);
        return;
    }

    raw_spin_unlock(&rnp->lock);  /*leave irqs disabled*/

  /*针对多层 struct rcu_node 结构的情况，把每一层 struct rcu_node 结构都重新更新
  为初始值*/
    rcu_for_each_node_breadth_first(rsp, rnp) {
        raw_spin_lock(&rnp->lock);  /*irqs already disabled*/
        …
      //更新 qmask 初始值
        rnp->qsmask = rnp->qsmaskinit;
      //更新 GP 号
        rnp->gpnum = rsp->gpnum;
      //更新 completed 号
        rnp->completed = rsp->completed;
      /*对与当前处理同属第一层 struct rcu_node 的处理器更新其 nxttail 数组指针*/
        if (rnp == rdp->mynode)
            rcu_start_gp_per_cpu(rsp, rnp, rdp);
        …
        raw_spin_unlock(&rnp->lock);   /*irqs remain disabled*/
    }

    …
}
```

10.3　RCU 函数的执行

（1）在每颗处理器 Tick 的时候，会检查当前处理器上是否有 RCU pending，如果成立将导致 Raise RCU_SOFTIRQ。

（2）RCU_SOFTIRQ 软中断中会调用 static void __rcu_process_callbacks(struct rcu_state *rsp, struct rcu_data *rdp)函数来检查。

```
static void __rcu_process_callbacks(struct rcu_state *rsp, struct rcu_data
*rdp)
{
     …
    /*针对本处理器，检查自己的 RCU_WAIT_TAIL 队列等待的 grace time 是否到期，如果已
    到期则将各队列依次向前调整*/
    rcu_process_gp_end(rsp, rdp);
    //从这里逐层检查各 CPU
    rcu_check_quiescent_state(rsp, rdp);
    …
    //当前处理器的 RCU_DONE_TAIL 队列不空，说明有过了 GP 的 RCU 函数需要运行
    if (cpu_has_callbacks_ready_to_invoke(rdp))
        invoke_rcu_callbacks(rsp, rdp);
}
//该函数软中断环境中执行
static void invoke_rcu_callbacks(struct rcu_state *rsp, struct rcu_data
*rdp)
{
    …
    //因为处在软中断环境，只有有限级较高的 RCU 函数才放在这里执行
    if (likely(!rsp->boost)) {
        rcu_do_batch(rsp, rdp);
        return;
    }
    /*对于普通的RCU函数在内核线程rcu_cpu_kthread中执行，这里唤醒rcu_cpu_kthread
    内核线程*/
    invoke_rcu_callbacks_kthread();
}
```

（3）rcu_cpu_kthread 内核线程的主体如下：

```
static void rcu_kthread_do_work(void)
{   …
    //该函数依次取出挂在 struct rcu_data 结构的 struct rcu_head 队列，依次执行
    rcu_do_batch(&rcu_sched_state, &__get_cpu_var(rcu_sched_data));
    …
}
```

第 11 章　MMC Driver

这是本书分析的唯一一个驱动子系统。本书原计划不想涉及驱动，但这是笔者的心结，因为笔者觉得只有 Block 层通了以后，内核横向架构才算是得到了贯通的分析：虚拟内存、VFS、文件系统、BLOCK 和块设备驱动。

11.1　MMC Driver

MMC Driver 有以下三个方面。

（1）MMC Host 即 MMC 控制器，与具体芯片有关，在 host 目录下可以发现不同 SOC 的 MMC 控制器。这里涉及具体芯片操作，对于架构分析意义不大，本文略去 MMC 控制器层面的分析。

（2）MMC 协议，即运行在 MMC 控制器上的协议，一个 MMC 控制器可能支持 SDIO、SD、MMC 等不同设备。针对不同设备在 Core 目录中有着不同实现。

（3）块设备，即内核意义的块设备，这里将底层的 MMC 设备抽象层上文提到抽象块设备，并接驳到内核块设备层。

11.1.1　MMC 协议层

MMC 控制器上电后，MMC 协议层要做第一件事是依次通过 MMC 控制器检测是否有满足自己的协议规范的设备。

```
//MMC 控制器上电后，驱动调用该函数检测 MMC 设备
static int mmc_rescan_try_freq(struct mmc_host *host, unsigned freq)
{
    ...
    /*SDIO 设备检测 */
    if (!mmc_attach_sdio(host))
        return 0;
    /*SD 设备检测 */
    if (!mmc_attach_sd(host))
        return 0;
    /*EMMC 设备检测 */
    if (!mmc_attach_mmc(host))
        return 0;
```

```
    mmc_power_off(host);
    return -EIO;
}


/*
 以 EMMC 为例进一步分析，该函数关于 EMMC 协议操作的部分略去，着重分析其与 EMMC 设备的检
 测与注册
 */
int mmc_attach_mmc(struct mmc_host *host)
{
    …
    /*
     操作 emmc host 检测 emmc 设备，若检测到 emmc 设备，则生成 struct mmc_card 并挂
     到 struct mmc_host 下
     */
    err = mmc_init_card(host, host->ocr, NULL);
    …
   /*将 struct mmc_card 注册到驱动框架，将触发标准的 device 和 driver 匹配*/
    err = mmc_add_card(host->card);
   …
}
```

11.1.2　MMC 块设备

在 struct mmc_card 注册时将匹配其对应的 EMMC 块设备驱动函数，其实现位于 driver/mmc/card/block.c，提供的驱动列表如下：

```
static struct mmc_driver mmc_driver = {
    .drv         = {
        .name    = "mmcblk",
    },
    .probe       = mmc_blk_probe,
    .remove      = mmc_blk_remove,
    .suspend     = mmc_blk_suspend,
    .resume      = mmc_blk_resume,
};
通过 mmc_bus_type 匹配名为 mmcblkdevice

//探测函数是 MMC 驱动的骨架
static int mmc_blk_probe(struct mmc_card *card)
{
    struct mmc_blk_data *md, *part_md;
    …
   /*创建 struct gendisk 和 struct request_queue 关键函数，MMC 块设备的基础设施
```

```
    都是这里构建，下文详细分析*/
    md = mmc_blk_alloc(card);
    if (IS_ERR(md))
        return PTR_ERR(md);
    //启动 host 操作，设置扇区大小
    err = mmc_blk_set_blksize(md, card);
    if (err)
        goto out;

    …
   /*对于 EMMC 卡，其头部有两个 boot partition，这里为其创建 struct gendisk，尽管
   在 emmc spec 是 partition 的概念，在 Linux 里将其当作独立的块设备来对盘，其主设备
   号为 0xb3，次设备号分别从 0x20，0x40 开始*/
    if (mmc_blk_alloc_parts(card, md))
        goto out;

    mmc_set_drvdata(card, md);
   //硬件错误修补
    mmc_fixup_device(card, blk_fixups);
   /*向 block 层注册*/
    if (mmc_add_disk(md))
        goto out;

   /*将 boot partition 对应上独立的块设备向 block 注册*/
    list_for_each_entry(part_md, &md->part, part) {
        if (mmc_add_disk(part_md))
            goto out;
    }
…
}

//MMC 驱动的基础设施是这里创建的
static struct mmc_blk_data *mmc_blk_alloc_req(struct mmc_card *card,
                            struct device *parent,
                            sector_t size,
                            bool default_ro,
                            const char *subname)
{
    struct mmc_blk_data *md;
    int devidx, ret;

    …
   //struct mmc_blk_data 对应一张 eMMC 或者 SD 卡，MMC 驱动 block 层使用
    md = kzalloc(sizeof(struct mmc_blk_data), GFP_KERNEL);
    if (!md) {
```

```
        ret = -ENOMEM;
        goto out;
    }


    //分配内核 bock 层使用 struct gendisk 结构，perdev_minors 是允许的分区数
    md->disk = alloc_disk(perdev_minors);
    …
    //struct mmc_queue 的创建和关键动作将后面详细分析
    ret = mmc_init_queue(&md->queue, card, &md->lock, subname);
    if (ret)
        goto err_putdisk;

    //初始化 struct gendisk 结构的详细信息
    md->queue.issue_fn = mmc_blk_issue_rq;
    md->queue.data = md;

    md->disk->major = MMC_BLOCK_MAJOR;
    md->disk->first_minor = devidx * perdev_minors;
    //disk 设备文件操作函数
    md->disk->fops = &mmc_bdops;
    md->disk->private_data = md;
    /*struct request_queue 由 MMC 驱动提供，意味着 MMC block 设备有自己的 struct
    backing_dev_info 结构*/
    md->disk->queue = md->queue.queue;
    md->disk->driverfs_dev = parent;
    set_disk_ro(md->disk, md->read_only || default_ro);
    md->disk->flags = GENHD_FL_EXT_DEVT;


    //设置 block 层扇区大小
    blk_queue_logical_block_size(md->queue.queue, 512);
    //设置 block 层设备容量大小
    set_capacity(md->disk, size);
…
}

int mmc_init_queue(struct mmc_queue *mq, struct mmc_card *card,
          spinlock_t *lock, const char *subname)
{
    …

    mq->card = card;
    /*创建 struct request_queue 其 request_fn_proc 函数为 mmc_request。struct
```

```
    backing_dev_info 初始化在这里完成*/
     mq->queue = blk_init_queue(mmc_request, lock);
     …
    //struct request_queue 的 prep_rq_fn;函数为 mmc_prep_request
     blk_queue_prep_rq(mq->queue, mmc_prep_request);

…
//负责将 block 提交的操作转发给 mmc host
mq->thread = kthread_run(mmc_queue_thread, mq, "mmcqd/%d%s",
        host->index, subname ? subname : "");
…
}


}
```

11.2 开源手机 U8836D（MT6577）分区的实现

这里的分区，不是操作系统的分区概念，也和架构没有太大关系，关心系统架构的读者可以直接略过此节。

本节描述的是 EMMC 上的 Android 镜像的布局，对于 MTK 系列 Android 手机来讲 hancking 与 ROM 的烧写有着较大的实用意义。

从华为开源的代码里可以看出，内核是通过读取写在 EMMC user 分区里的分区表来获得系统分区信息的，而分区的信息位置是固定的，从 1024 开始，起始位置签名为 0x50547631 的扇区；或者从 1024+ 2048 开始，起始位置签名为 0x4D505431 的扇区。这个位置应该是 MTK 的烧写软件与 bootm 约定好的，因为从 log 里看 preloader 也在里面，而 bootm 必取 preloader。也许 mtk 烧写软件将 preloader 同时写入 EMMC 的 boot 分区，但是这样又限制的 EMMC 版本，为了支持庞大的 EMMC 供应商列表，MTK 应该不会这样做。

```
static int load_exist_part_tab_emmc(u8 * buf)
{
        int reval = ERR_NO_EXIST;

        int i,j;
        int len=0;
        //读入内存的分区表的指针
        char *buf_p;
        //从这个位置开始找分 magic 为 0x50547631 的分区表
        loff_t pt_start = 1024;
        //从这个位置开始找分 magic 为 0x4D505431 的分区表
        loff_t mpt_start = pt_start + 2048;
        loff_t pt_addr = 0;
```

```
int pn_per_pmt = 0;
//分区表大小为 2K，分区表的最后也有签名
int per_pmt_size = 2048;

pt_resident32 *lastest_part32;
int blk_size = 512;
//每次读取的长度
int read_size = 16*1024;//8192;//4096;
char *page_buf = NULL;
char *backup_buf = NULL;
…
//读取信息暂存在 page_buf
page_buf = kmalloc(read_size, GFP_KERNEL);
…
/*从 1024 开始，找起始位置签名为 0x50547631 的扇区*/
for(i=0;i<CFG_EMMC_PMT_SIZE/read_size;i++)
{
    buf_p = page_buf;
    //启动 emmc 读
    reval = eMMC_rw_x(pt_start + i*read_size,(u32*)page_
    buf,0,0,read_size,1,USER);
    …
    //逐个扇区找签名
    for(j=0;j<read_size/blk_size;j++){
        //条件成立，则表示分区表的签名找到
        if(is_valid_pt(buf_p)){

            //当次读取的内容没有把分区表全部读进来
            if((read_size-j*blk_size) < per_pmt_size){
                len = read_size- j*blk_size;
                //把已经读进来的分区表放入 backup_buf
                memcpy(backup_buf,&buf_p[PT_SIG_SIZE],len-PT_
                SIG_SIZE);
                //再次启动 EMMC 读，把剩下分区表读进来
                reval = eMMC_rw_x(pt_start + (i+1)*read_size,(u32*)
                page_buf,0,0,per_pmt_size,1,USER);
                …
                /*检查分区表的尾部签名，如果仍然存在签名，说明分区表还有另
                外一部分，长度为 per_pmt_size*/
                if(is_valid_pt(&page_buf[per_pmt_size-4-len])){
                    //记录下分区表的位置
                    pt_addr = pt_start + i*read_size+j*blk_size;
                    pi.pt_has_space = 1;
                    reval=DM_ERR_OK;
                    goto find;
```

```
                    }

                }else{
                    //这种情况是，一次即把分区表全都读进来，检查尾部签名
                    if(is_valid_pt(&buf_p[per_pmt_size-PT_SIG_SIZE])){

                        …
                    //记录分区表另一个部分的位置
                        pt_addr = pt_start + i*read_size+j*blk_size;
                        pi.pt_has_space = 1;
                        reval=DM_ERR_OK;
                        goto find;
                    }
                }
                break;
            }
        buf_p += blk_size;
      }
    }
    //从 1024 开始的位置没有搜索到分区表
    if(i == CFG_EMMC_PMT_SIZE/read_size)
    {
        pi.pt_has_space = 0;
        /*从 mpt_start 位置搜索分区表，方法同上参见上文，大多数情况下不会走到这里*/
        for(i=0;i<CFG_EMMC_PMT_SIZE/read_size;i++){
        …
        }
}
    //没有找到分区表的情况
    if(i == CFG_EMMC_PMT_SIZE/read_size){
    …
    reval = ERR_NO_EXIST;
    }
   goto end;
//分区表已被读到内存中 backup_buf[]，做进一步处理
find:
  //如果还有另一部分，那么在当前这个 per_pmt_size 的尾部有标准位指示
  pi.pt_next = (backup_buf[per_pmt_size-11]>>4)&0x0F;
   pi.sequencenumber = backup_buf[per_pmt_size-12];
   …
   //继续读入另一部分分区表，这种情况很少见
   if(pi.pt_next == 0x1){
   //pt_addr 已经记录了分区另一部分在 EMMC 上的位置，启动读取操作
        reval = eMMC_rw_x(pt_addr+per_pmt_size,(u32*)page_buf,0,
        0,per_pmt_size,1,USER);
```

```
        …
    //再次检查第二部分分区表的签名
    if((is_valid_pt(page_buf)&&is_valid_pt(&page_buf[per_pmt_size-4]))|
    |(is_valid_mpt(page_buf)&&is_valid_mpt(&page_buf[per_pmt_size-4]))){
                    pt_next = 1;
                    //把读进来分区表信息整理到 backup_buf
                    if(emmc_size<0x100000000ULL){
    memcpy(&backup_buf[pn_per_pmt*sizeof(pt_resident32)],&page_buf[4],
    per_pmt_size-8);
                        }else{
    memcpy(&backup_buf[pn_per_pmt*sizeof(pt_resident)],&page_buf[4],per_
    pmt_size-8);
                            }
                }else{
                    printk("can not find next pt, error\n");
                    }
    }
    if(emmc_size<0x100000000ULL){ //32bit
        …
        //backup_buf 信息倒腾到 lastest_part32
        memcpy(lastest_part32,backup_buf,PART_MAX_COUNT*sizeof(pt_
        resident32));
        //逐项整理 lastest_part32 信息待内核使用
        for(i=0;i<PART_MAX_COUNT;i++)
        {
            if(lastest_part32[i].name[0]!=0x00){
                memcpy(lastest_part[i].name,lastest_part32[i].name,MAX_
                PARTITION_NAME_LEN);
                lastest_part[i].size= lastest_part32[i].size;
                lastest_part[i].offset= lastest_part32[i].offset;
                lastest_part[i].mask_flags=     lastest_part32[i]
                .mask_flags;
            }
        }
    }else{
        …
    }
   …
}
```

第 12 章　内核配置系统及内核调试

12.1　Conf

在做 ARM BSP 移植的时候经常会弄错内核模块的依赖关系。作为工具性质的最后一章内核部分，本节分析内核配置系统的原理，以供参考。

12.1.1　Kconfig 元素

Kconfig 最基本的元素是字符串，由 struct symbol 表示，是组成其他元素的最基本单元。

Kconfig 的结构就是 menu 的层级结构，由三种类型的 menu 的层级构成。

（1）第一级是 menu，其基本结构是 menu…endmenu。

在 arch/arm/kconfig 文件中，有个处理器架构选择就是 menu：

```
menu "System Type"
config MMU
    bool "MMU-based Paged Memory Management Support"
    default y
    …
choice
    prompt "ARM system type"
endmenu
```

该 menu 的菜单项由 config MMU、choice 构成。

（2）第二级是 Choice，其基本结构是 choice … endchoice，例如：同样在 arch/arm/kconfig 文件中，对 SOC 架构选择就是 Choice。

```
choice
    prompt "ARM system type"
    default ARCH_VERSATILE
…
endchoice

config ARCH_SOCFPGA
    bool "Altera SOCFPGA family"
    select ARCH_WANT_OPTIONAL_GPIOLIB
    select ARM_AMBA
```

```
    …
config ARCH_REALVIEW
    bool "ARM Ltd. RealView family"
    select ARM_AMBA
    select CLKDEV_LOOKUP
    …

config ARCH_VERSATILE
    bool "ARM Ltd. Versatile family"
    select ARM_AMBA
    …
…
endchoice
```

该 menu 的菜单项由不同的 config XXXX 构成。

（3）第三级是 config，这是最低级的菜单，不包含菜单项，但是有相应的属性，如：

```
config ARCH_S5PV210
    bool "Samsung S5PV210/S5PC110"
    select CPU_V7
    select ARCH_SPARSEMEM_ENABLE
    select ARCH_HAS_HOLES_MEMORYMODEL
    select GENERIC_GPIO
    select HAVE_CLK
    select CLKDEV_LOOKUP
    …
    help
      Samsung S5PV210/S5PC110 series based systems
```

12.1.2　Kconfig 分析

Kconfig 的分析很简单，如下：

（1）遇到任何非菜单元素都是该 current_menu 的属性。

（2）遇到菜单元素时，生成并进入新菜单，因为任何时候都有一个当前 menu，由 struct menu *current_menu 表示。上一级菜单记录为当前菜单的成员变量 struct menu *parent;：menu->parent = current_menu;。

（3）当前 menu 结束，struct menu *current_menu 恢复为本级菜单的父母菜单。

```
enum yytokentype {
    T_MAINMENU = 258,
    T_MENU = 259,
    T_ENDMENU = 260,
    T_SOURCE = 261,
    T_CHOICE = 262,
```

```
    …
}
```

Choice 的处理：Choice 其实就是一种 menu，遇到 Choice 时处理与 menu 处理相似。

```
case 51:
    {
    /*为这个 Choice 名字生成 struct symbol 结构*/
    struct symbol *sym = sym_lookup((yyvsp[(2) - (3)].string), SYMBOL_
    CHOICE);
    sym->flags |= SYMBOL_AUTO;
    /*为这个 choice 生成 struct menu 结构*/
    menu_add_entry(sym);
   …
    }
```

遇到 endchoice，代码如下：

```
  case 53:
    {
    if (zconf_endtoken((yyvsp[(1) - (1)].id), T_CHOICE, T_ENDCHOICE)) {
        /*把当前 menu 恢复到上一级 menu*/
        menu_end_menu();
    }
```

12.2 内 核 调 试

内核调试手段是一个广受争议的话题。笔者认为，内核调试的唯一有效手段是运行时的内核纯内存操作的 LOG。

对于内核调试，常用的调试应用程序的一些手段如断点、单步跟踪、修改内存等做法一律无效。这些调试手段有两种实现：一种是像 KGBD 这种靠在内核代码里添加代码，再从一个外部机器来控制；另外一种是用在线仿真器如 trace32、DS-5（realview）来控制处理器，再用台机器运行调试 UI、控制在线仿真器。这些手段虽然可以解决一些直观易发现的 bug，但是这些调试手段自身却破坏了内核原有的状态。

如内核断点企图让内核停下来以观察内核状态，但是内核实际运行的时候怎么可能停下来？当前线程不停向前走，各种中断噼里啪啦进来，触发各种服务例程，当前线程不断被打断、被抢占、被恢复……这才是真正的内核运行状态。如果忽略这些状况，强行钳住处理器，或者甚至靠添加调试代码来造成内核停止、单步了的现象都是假象，是实际运行时不可能存在的。

内核不是被驯服的应用线程，内核调试唯一有效手段就是在内核代码里手工加入修改掉设备操作的 printk，再想法输出内核运行信息，然后通过分析这些运行信息来调试代码。

未经修改的 printk 并不是在内核任何地方都能用，printk 本身要操作 console，需要操

作 semaphore 而且会引入中断。所以需要将 printk 的 console 操作去掉，让 printk 不去操作硬件，不去 down semaphore。这样 printk 就是一个纯操作内存的函数，就能够在内核的任何一个地方使用了。

其实这样还会对内核造成影响，就是 printk 也是需要花时间的，处理器要把 printk 跑掉。在较弱的处理器上尤其要注意，在 ARM7 上调试的时候笔者曾发现，有时驱动里 printk 加的多了，外设来不及响应，最后发现是跑 printk 的时间太长了。现在的处理器 CA8/CA9 跑 printk 已经非常快了，不过随着 Linux 内核复杂度的飙升，任何时间都不能忽视处理器跑内核代码花费的时间对系统的影响。

说到这里，还有个问题没有解决，pintk 把内核信息打印到内存了，但是怎么才能把这些内存里的信息捞出来呢？方法有很多，笔者最喜欢以下两种方式。

（1）通过在线仿真器把 LOGBUF 里的内存荡出来，市面上的在线仿真器中，不论是独步天下的 trace32、DS-5 还是山寨货，所有的仿真器都有下载内存功能，只是下载速度快慢的问题。仿真器调试内核适合比较生猛的情况，比如把 Linux 移植到全新处理器架构下，前几年笔者有个项目把 Linux 2.6 移植到某 DSP 上，刚开始 Arch 下什么都还没有，只能靠仿真器把内核甩下去，运行一定时间再把 LOGBUF 荡出来。但是这种方式有个弊端，就是需要开一个比较大的 LOGBUF，因为要在最后一次性把内存荡出来，一个 4M 的 LOGBUF 还经常溢出，荡一次内存往往要七八分钟。

（2）第二种方式适合在一个基本稳定但又需要追踪比较罕见的 BUG 的时候，往往需要荡出来数十 M 的 LOG 信息，这样需要既能 pintk 到内存，又能同时往外输出。这种情况下，笔者通常的作法如下。

① 保持 pintk 不变，在适当时机、适当地方执行 printk，这时 printk 会把整个 LOGBUF 信息输出。

② 再做个函数 senix_printk，该函数基于 printk 改造，把 printk 操作 console 相关地方去掉，该函数依旧输出到 LOGBUF，因为有 logbuf_lock 锁存在，所以不会打扰原始 ptinrk 的输出。

这样方式的前提是内核基本 OK，串口能输出，printk 的时机要在 LOGBUF 满之前得到执行，否则会溢出。详细分析见下文。

12.2.1　senix_printk

本节分析常用的内核调试方法，阐述思路供大家参考、指正。

```
/*senix_printk 与 printk 参数定义完全一样，且在 printk.h 里暴露给内核使用*/
asmlinkage int senix_printk(const char *fmt, …)
{
    va_list args;
    int r;
    …
    //获取参数
    va_start(args, fmt);
```

```
    //实际工作换成 senix_vprintk
    r = senix_vprintk(fmt, args);
    va_end(args);

    return r;
}

//senix_vprintk 从 vprintk 改造而来
asmlinkage int senix_vprintk(const char *fmt, va_list args)
{
    int printed_len = 0;
    int current_log_level = default_message_loglevel;
    unsigned long flags;
    int this_cpu;
    char *p;
    size_t plen;
    char special;
…
/*把当前处理器状态寄存器值存入 flags，并将状态寄存器中断位置位，禁止中断。这里关中断是
为了防止在处理器执行下面代码的时候，当前处理器上的中断窜进来，导致当前处理器跑到这里。
printk 通过 spin_lock(&logbuf_lock);来保证 LOGBUF 的访问，spin_lock(&logbuf_
lock)就像一扇门，在多颗处理器狭路相逢的时候挡得住别的处理器，而在自己通过狭路的时候
spin_unlock(&logbuf_lock)再打开这扇门，让别的处理器通过狭路。但是当前处理器处在狭
路的时候，处理器被抢占，又重新进入狭路之前，等待狭路之门打开，但是这时却没有人能够打开狭路
之门了。唯一能够抢占狭路上的 CPU 只有中断，所以这里关掉中断，防止这种极端事件产生*/
    raw_local_irq_save(flags);
    //当前 CPU 号
    this_cpu = smp_processor_id();

/*如果 printk_cpu 是当前 CPU，那说明 printk 被不正常重入了，这与抢占不同，这是处理器
跑乱了*/
    if (unlikely(printk_cpu == this_cpu)) {

        /*进入到这里有两种可能：(1) 在 pintk 是内核崩溃，而内核崩溃前又要把信息打出去，
        如果崩溃很频繁，比如新处理器移植的初期，就得用仿真器了，console 并不把全部信息
        输出出去：(2) printk 时又递归调用了 printk，printk 代码不止是大家在书里看到
        的这些，在 console_unlock();里有一大坨驱动相关的代码，里面有可能调用 printk*/

        if (!oops_in_progress) {
            recursion_bug = 1;
            goto out_restore_irqs;
        }
        zap_locks();
    }
```

```
 lockdep_off();
//狭路之门，关闭
 spin_lock(&logbuf_lock);
//printk_cpu 为当前 cpu
 printk_cpu = this_CPU;
 …
 printed_len += vscnprintf(printk_buf + printed_len,
               sizeof(printk_buf) - printed_len, fmt, args);

/*printk_buf 已经被放入要输出的格式及参数信息*/

 p = printk_buf;

 /*正常情况下，printk_buf 的前三位 P[0] ,P[1], P[2]为 <L>，L 是内核定义的
 KERN_EMERG -KERN_DEBUG, KERN_DEFAULT, KERN_CONT，这里 current_log_leve
 记录 L 的信息，如果 L 为 KERN_DEFAULT, KERN_CON，speical 记录其信息 */
 plen = log_prefix(p, &current_log_level, &special);
 if (plen) {
     p += plen;

     switch (special) {
     case 'c': /*KERN_CON 表示继续前面一行输出*/
         plen = 0;
         break; /*break，不产生以后*/
     case 'd': /*KERN_DEFAULT, 表示新起一行 */
         plen = 0;
     default:
         if (!new_text_line) {
           //只要没有指定 KERN_CON，就是新一行，这里输出到 LOGBUF 中
             emit_log_char('\n');
             new_text_line = 1;
         }
     }
 }
 /*依次处理剩下的输出信息，将其复制到 LOGBUF */
 for (; *p; p++) {
 //处理每一行
     if (new_text_line) {
         new_text_line = 0;
         …
     /*时间戳是在这里取得 */
         if (printk_time) {
             /*Add the current time stamp */
             char tbuf[50], *tp;
             unsigned tlen;
```

```
                unsigned long long t;
                unsigned long nanosec_rem;
            /*这里直接从硬件计时器取值，但是取得时间跟打印信息的时间有个偏差，而且
            从调用 printk 到走到关中断之前是有可能被线程或者中断抢占，如果要获得精度
            较高的时间需要自己在代码里取*/
                t = cpu_clock(printk_cpu);
                …
                for (tp = tbuf; tp < tbuf + tlen; tp++)
                //把时间送到 LOGBUG
                    emit_log_char(*tp);
                    printed_len += tlen;
            }
            …
        if (!*p)
                break;
        }//每一行开头的处理器到此结束
        //把这一行的信息输出到 LOGBUF
        emit_log_char(*p);
        //遇到\n 时另起一行
        if (*p == '\n')
            new_text_line = 1;
    }
    //关键的改动在这里，删掉 console 相关操作
    //if (console_trylock_for_printk(this_cpu))
    //  console_unlock();
    //当前处理器离开了 pintk，printk_cpu 被置为 UINT_MAX
    printk_cpu = UINT_MAX;
    //狭路之门开启
    spin_unlock(&logbuf_lock);

    lockdep_on();
out_restore_irqs:
    /*恢复中断状态，这里不是打开中断，而是把进入 printk 之前的处理器状态寄存器恢复*/
    raw_local_irq_restore(flags);
    …
    return printed_len;
}
```

12.2.2 LOG_BUF

LOG_BUF 是用来放置 printk log 信息的地方，有两种实现方式。

一种直接在内核里定义一个大数组：

```
static char __log_buf[__LOG_BUF_LEN];
static char *log_buf = __log_buf;
```

一种方式是在内核参数里指定其大小：log_buf_len，内核分析到该参数用 static unsigned long __initdata new_log_buf_len;记录其大小，然后在内核启动时调用 void __init setup_log_buf(int early)来显示的创建 LOG_BUF。

```
void __init setup_log_buf(int early)
{   …
   //如果启动参数指定 LOG_BUF 的大小，那么内核会在 new_log_buf_len 记录这个长度
    if (!new_log_buf_len)
        return;
  /*内核动态的创建 LOG_BUF 的基础是内存分配，正常的内存分配体系是 buddy system，这是
  一串串 4K 整数倍的内存块;在这之前的是bootmem，这是一片位图，每位代表一页;而在bootmem
  是从 bootloader 里收集的内存分布信息。正常情况下在 bootmem 后再动态创建 LOG_BUF，
  但是有些系统上比较着急，要尽快分配比较大的 LOG_BUF，所以这就是 early 参数的意义*/
    if (early) {
        unsigned long mem;
        //bootmem 还没建立起来，只好从 memblock 分配内存
        mem = memblock_alloc(new_log_buf_len, PAGE_SIZE);
        if (mem == MEMBLOCK_ERROR)
            return;
        //算出分配出来的内存对于的虚拟地址
        new_log_buf = __va(mem);
    } else {
       /*从 bootmem 里分配内存比较从容了。new_log_buf 直接返回的就是虚拟地址*/
        new_log_buf = alloc_bootmem_nopanic(new_log_buf_len);
    }

    …
   /*分配好了新的 LOG_BUF，下面要进行 LOG_BUF 的切换，关中断、锁住 logbuf_lock，只
   有 CPU0 在跑，这里只可能有中断来打扰，只关中断即可，不需要锁住 logbuf_lock*/
    spin_lock_irqsave(&logbuf_lock, flags);
    /*log_buf 数组是内核静态定义的，不管动不动态分配，它就在那里，现在要把静态数组的
    log_buf 里的数据倒到新分配的 LOG_BUF 里，并且把相关指针切换过来*/
    log_buf_len = new_log_buf_len;
    //log_buf 指向新分配内存
    log_buf = new_log_buf;
    new_log_buf_len = 0;
    free = __LOG_BUF_LEN - log_end;
   //con_start, log_start 是 LOG_BUF 的两个机制，见下文
    offset = start = min(con_start, log_start);
    dest_idx = 0;
   //把 log_buf 里的数据倒到新分配的 LOG_BUF，LOG_BUF 已指向新 BUF
    while (start != log_end) {
        unsigned log_idx_mask = start & (__LOG_BUF_LEN - 1);

        log_buf[dest_idx] = __log_buf[log_idx_mask];
```

```
            start++;
            dest_idx++;
        }
        /*静态数组的 log_buf 里的数据倒进来了，等于 LOG_BUF 内容增加，自然要把指针往后推
        一推*/
        log_start -= offset;
        con_start -= offset;
        log_end -= offset;
        //解锁，恢复先前的中断状态
        spin_unlock_irqrestore(&logbuf_lock, flags);
    }
```

LOG_BUF 的运作主要是靠以下三个指针来实现的。

（1）log_start：Linux 系统上 sysklogd 从这里提取 log。

（2）con_start：控制台从这里输出 log 信息。

（3）log_end：log 信息的结束地址。

```
    //该函数是向 LOG_BUF 写 LOG 信息的唯一入口
    static void emit_log_char(char c)
    {
        //log_end 是 LOG 的最后地址，也是第一个空闲内存，写入
        LOG_BUF(log_end) = c;
        //log_end 指针向后推移
        log_end++;
        //循环 BUF，若 log_end 追上了 log_start，log_start 向后串
        if (log_end - log_start > log_buf_len)
            log_start = log_end - log_buf_len;
        //循环 BUF，若 log_end 追上了 con_start，con_start 向后串
        if (log_end - con_start > log_buf_len)
            con_start = log_end - log_buf_len;
        if (logged_chars < log_buf_len)
            logged_chars++;
    }
```

在每次调用 printk 时，都会调用 void console_unlock(void)。该函数是 LOG_BUF 信息输出的关键动作。在进入该函数之前，需要拿到 console semaphore。

```
    void console_unlock(void)
    {
        …
    /*该函数主要对 console 操作，如果 console 不可用，也就没有可以继续下去的意义了，释放
    semaphore，返回*/
        if (console_suspended) {
            up(&console_sem);
            return;
        }
```

```
        console_may_schedule = 0;
        for ( ; ; ) {
            //关中断，锁 logbuf_lock，这里用得很恰当
            spin_lock_irqsave(&logbuf_lock, flags);
            //先做件好事，看看是否叫醒 sysklogd
            wake_klogd |= log_start - log_end;
            /*检查自己是否有工作要做，如果 con_start == log_end 说明没有需要 console
            输出的信息*/
            if (con_start == log_end)
                break;          /*Nothing to print */
            //有活要干，要把从 con_start 到 log_end 的 log 信息从 console 里送出去
            _con_start = con_start;
            _log_end = log_end;
            con_start = log_end;        /*Flush */
            /*解开 logbuf_lock，把别的处理器放进来，这时 con_start == log_end 已成立，
            不担心别的处理器来瞎折腾*/
            spin_unlock(&logbuf_lock);
            //调用 console 驱动把 log 信息送出去
            call_console_drivers(_con_start, _log_end);
            /*直到这里才开中断，这就能保证如果 console 驱动不开中断，可以得到个干净的输出*/
            local_irq_restore(flags);
        }
        console_locked = 0;
        //释放控制台
        up(&console_sem);
        //这里对应上面的 break 情况
        spin_unlock_irqrestore(&logbuf_lock, flags);
      /*sysklogd 一般通过 syslog 趴在 log_wait 上，叫醒 sysklogd。一个系统应该尽量干净，
      在嵌入式系统或者 Android，这个机制最好不要用*/
        if (wake_klogd)
            wake_up_klogd();
    }
```

Printk 表面上是个简单的打印函数，但是其下是较复杂的 console 驱动的体系，而更关键的是 printk 被用在内核各种情况下，这使得复杂度被急剧放大。笔者最近遇到的一个教训是：SMP 系统下，CPU1 的启动过程会丢失 LOG_BUF 的内容。原因如下：printk 调用控制台驱动经过如下函数。

```
    //注意黑体部分代码
    static void __call_console_drivers(unsigned start, unsigned end)
    {
        for_each_console(con) {
            ...
            if (conenable && con->write &&
                //检查当前 CPU 是否在线
```

```
                (cpu_online(smp_processor_id()) ||
            //恰巧该平台下的 console flags 是 CON_PRINTBUFFER
                (con->flags & CON_ANYTIME)))
            con->write(con, &LOG_BUF(start), end - start);
    }
}
```

该函数在确认当前 CPU 在线后才会进入 console 的写函数，在处理器 online 被置位之前调用 printk，将出现 LOG_BUF 的丢失，无论是通过 senix_printk 写入还是通过 printk 的 LOG 信息都被冲掉。因为从处理器启动到 set_cpu_online(cpu, true);这段时间，cpu_online(smp_processor_id()都为否，但是这时控制台的 con_start 都被指向了 log_end，于是 LOG_BUF 信息被丢失。

下篇　Dalvik 与 Android 用户态源码分析

Android 用户态== Dalvik 虚拟机+Binder+功能类库。

第 13 章　内　　存

作为系统中最简单也是最复杂的系统，内存管理似乎是一个永远也说不完的话题。在内核部分可以看到无论虚拟内存还是物理内存，其基本管理单位都是以页为单位，对于 32 位 ARM Android 系统中默认为 4K。在 Android 用户层生活着 Dalvik 虚拟机系统与应用进程以及系统Deamon，本章分析它们的虚拟内存是如何使用的以及其与4K内存页面的关系。

13.1　Dalvik 内存管理

Dalvik的内存分配回收机制直接建构在Bionic C库的内存管理之上，物理上属于Bionic 的一部分，包括内存分配与回收两个方面。前者接驳于 Dalvik 的对象分配机制，后者接驳与 Dalvik 垃圾回收机制。

用户层的虚拟使用，有堆和栈两种方式，栈的实现非常简单，连续的向着某个方向增长或弹出，只要阅读完内核部分的虚拟内存管理，就明白了二进制线程栈的机理，Java 栈的分析请参见解释器部分，这里的分析对象是堆机制。

13.1.1　虚拟内存分配

用户层虚拟内存的分配前提是该段待分配虚拟内存地址被内核确认为有效，即通过 map 机制在内核层面的进程虚拟内存管理结构中分配出该段内存的地址。然后用户层通过 malloc 在进行分割得到适合用户代码使用的更小虚拟内存段。但是这个分配内存段却不能保证其有着实际对应的物理内存。原因在于，物理内存都是成页分配并挂载到页表项的，通常以 4K 为单位。这个 4K 物理内存有着对应 4K 的虚拟内存地址，而物理内存的分配是在 4K 地址中第一次写操作时发生的。所认用户层 malloc 出来的内存段所在 4K 虚拟内存之前就被写访问过，则其 malloc 之初就有着对应的物理内存，否则这个虚拟内存只是一个数字的概念。在有着 SWAP 机制的系统里，这些物理内存还存在随时被 shrink out 和 demand in 的可能。

在用户态每一次对 malloc 进行写操作以及第二次以后读操作时，都有可能导致当前线程的休眠，其原因在于有可能引起物理内存分配，而每次物理内存分配都有可能导致脏页回写、demand in 等导致休眠的动作。所以代码执行时间是要关注的，永远都要提防 malloc 出来的内存。

回到正题，malloc 的内存在 C 库里有着基本管理结构，这个结构尽管用编程看不到，但它就在进程空间存在着，该管理结构定义如下：

```
struct malloc_chunk {
  size_t             prev_foot;  /*Size of previous chunk (if free)*/
  size_t             head;       /*Size and inuse bits*/
  struct malloc_chunk* fd;       /*double links -- used only if free*/
  struct malloc_chunk* bk;
};
```

（1）成员变量 size_t prev_foot;在相邻的前面那块内存块空闲（free）时，记录前面那块内存块的大小。

（2）成员变量 size_t head，高 30 位记录本块内存的大小，第 0 位记录前面一块内存是否已经被分配（正在使用中），第 1 位记录本块内存是否已经被分配。

（3）成员变量 struct malloc_chunk* fd; struct malloc_chunk* bk;只在该块内存空闲时有用，是 smallbin 和 treebin 的节点

malloc 与 Dalvik 之间关系是，Dalvik 虚拟机本身以及系统的一些 native 函数和 deamon 使用 malloc，当 Dalvik 进行对象分配时，实质上是在虚拟内存中分配对象大小的内存空间，也使用该函数。

malloc 的实现有多种方式，在 Android Bionic 库里分配一块内存，首先从 smallbin 里找空闲的内存块，若没有找到则使用 smallbin 里更大的内存块分割。若还不满足就从 treebin 树里寻找适合节点，若还不行，就需要使用系统调用单独 map 大块内存或者扩大 struct malloc_state 尺寸。

```
//bionic c 库的 malloc 实现函数
void* mspace_malloc(mspace msp, size_t bytes) {
  mstate ms = (mstate)msp;
  …
  if (!PREACTION(ms)) {
    void* mem;
    size_t nb;

    // MAX_SMALL_REQUEST 默认值是 254
    if (bytes <= MAX_SMALL_REQUEST) {
      …
     //算出大小为 nb 的内存块对应的 smallbin 索引
      idx = small_index(nb);
        // mspace 的 smallmap 位图指出，是否有空闲的对应尺寸内存块
      smallbits = ms->smallmap >> idx;
      if ((smallbits & 0x3U) != 0) { /*Remainderless fit to a smallbin*/
        mchunkptr b, p;
        idx += ~smallbits & 1;       /*Uses next bin if idx empty*/
        //从 smallbin 链表里取出这个空闲的内存块
        b = smallbin_at(ms, idx);
        p = b->fd;
        assert(chunksize(p) == small_index2size(idx));
        unlink_first_small_chunk(ms, b, p, idx);
```

```
        /*将本块的 head 里面填上该内存块的大小，并且将下一个内存块的 head 变量里的
        PINUSE_BIT 置位，因为从下一个内存块的角度看来，前面的内存块处于已分配状态*/
        set_inuse_and_pinuse(ms, p, small_index2size(idx));
        mem = chunk2mem(p);
        check_malloced_chunk(ms, mem, nb);
        goto postaction;
      }
      else if (nb > ms->dvsize) {
        if (smallbits != 0) { /*Use chunk in next nonempty smallbin*/
        /*没有找到正好适合 nb 的空闲内存块，但是 smallbin 链表里还有更大的空闲内存待
        分配*/
          …
          binmap_t leastbit = least_bit(leftbits);
          compute_bit2idx(leastbit, i);
          //找到最适合的
          b = smallbin_at(ms, i);
          …
          unlink_first_small_chunk(ms, b, p, i);
          rsize = small_index2size(i) - nb;
          /*Fit here cannot be remainderless if 4byte sizes*/
          if (SIZE_T_SIZE != 4 && rsize < MIN_CHUNK_SIZE)
            set_inuse_and_pinuse(ms, p, small_index2size(i));
          else {
            set_size_and_pinuse_of_inuse_chunk(ms, p, nb);
            //r 是切完之后剩下的那一块
            r = chunk_plus_offset(p, nb);
            set_size_and_pinuse_of_free_chunk(r, rsize);
            //可怜的 r，被当成 dv 了
            replace_dv(ms, r, rsize);
          }
          …
        }
        else if (ms->treemap != 0 && (mem = tmalloc_small(ms, nb)) != 0) {
        //需要分配的 size 足够大，那就从 tree 里分配好了
          check_malloced_chunk(ms, mem, nb);
          goto postaction;
        }
      }
    }
  else if (bytes >= MAX_REQUEST)
    /*特大块内存使用情况较小，仅在某些架构不够优化的系统 deamon 里出现，需直接向内核 map
    出来*/
      nb = MAX_SIZE_T;
    else {
      nb = pad_request(bytes);
```

```
    …
  }

    //前面 smallbin 和 treebin 里都没有适合的，那就……又是挨砍最多的 dv
    if (nb <= ms->dvsize) {
    //从 dv 里砍一刀
    size_t rsize = ms->dvsize - nb;
    mchunkptr p = ms->dv;
    if (rsize >= MIN_CHUNK_SIZE) { /*split dv*/
      …
      set_size_and_pinuse_of_inuse_chunk(ms, p, nb);
    }
    else { /*exhaust dv*/
    //dv 彻底牺牲掉了
      size_t dvs = ms->dvsize;
      …
      set_inuse_and_pinuse(ms, p, dvs);
    }
    …
  }
  else if (nb < ms->topsize) { /*Split top*/
    //nb 足够大，又小于当前库存的 free 内存，从 top 里砍
    size_t rsize = ms->topsize -= nb;
    mchunkptr p = ms->top;
    …
    goto postaction;
  }
  //超大内存的分配，直接使用系统调用，获得大块虚拟地址
  mem = sys_alloc(ms, nb);
  …
 }
 …
}
```

随着应用的运行，对内存的需求越来越大，若导致虚拟内存空间的增长，需要向系统申请更多的虚拟内存。通常情况下，新申请的虚拟内存与原有虚拟内存地址是连续的。在申请完成之后需要将其管理结构与原有虚拟内存合并起来。

```
/*向系统申请虚拟内存*/
static void* sys_alloc(mstate m, size_t nb) {
  …
  /*对于大块虚拟内存（小于 mparams.mmap_threshold）的分配，可以通过三种方式进行，
  代码里有详细的注释，请读者自行阅读，这里不介绍*/

  if (tbase != CMFAIL) {
```

```
        …
        }
        else {
        / *已经分配了一块内存区域，下面要在这个 mstate 的 msegment seg;链表中寻找一块合
        适的区域与新分配的这块内存区域合并*/
          msegmentptr sp = &m->seg;
          while (sp != 0 && tbase != sp->base + sp->size)
            sp = sp->next;
          if (sp != 0 &&
              !is_extern_segment(sp) &&
              (sp->sflags & IS_MMAPPED_BIT) == mmap_flag &&
              segment_holds(sp, m->top)) { /*append*/
          /*找到一个既有的内存区域，该区域的结束地址是新分配区域的起始地址，合并之，在
          Android 系统中这是最常发生的情况，占了 system allocation 的 99%以上的概率*/
            sp->size += tsize;
            init_top(m, m->top, m->topsize + tsize);
          }
          else {
            …
            if (sp != 0 &&
                !is_extern_segment(sp) &&
                (sp->sflags & IS_MMAPPED_BIT) == mmap_flag) {
                /*找到一个既有的内存区域，该区域的起始地址就是新分配区域的结束地址，合并之*/
              char* oldbase = sp->base;
              sp->base = tbase;
              sp->size += tsize;
              return prepend_alloc(m, tbase, oldbase, nb);
            }
            Else
              /*没有任何一个既有内存区域与新分配区域相邻，那只好再加个新的 msegment*/
              add_segment(m, tbase, tsize, mmap_flag);
          }
        }

        if (nb < m->topsize) { /*Allocate from new or extended top space*/
          //真正的 nb 分配时刻
          size_t rsize = m->topsize -= nb;
          mchunkptr p = m->top;
          …
          return chunk2mem(p);
        }
      }
    …
    }
```

```
/*新的 segment，关于 segment 与 mstate 的分析，在 malloc.c 文件的头部有着详细的注释，
请读者自行阅读*/
static void add_segment(mstate m, char* tbase, size_t tsize, flag_t mmapped)
{
  …
 char* old_top = (char*)m->top;
  …
  /*mstate 的 top 指向新分配区域，因为既有的 top segment 满足不了 nb size 的分配要
  求，而新分配内存区域可以，这就是替换的原因*/
  init_top(m, (mchunkptr)tbase, tsize - TOP_FOOT_SIZE);
  /*把新的 segment 挂到 mstate 的 msegment seg;链表上，典型的链头操作*/
  …
  set_size_and_pinuse_of_inuse_chunk(m, sp, ssize);
  *ss = m->seg; /*Push current record*/
  //基址
  m->seg.base = tbase;
  …

  /*原有的 segmnet 根据大小被挂到 bin 链表或者 tree 中，供小内存使用*/
  if (csp != old_top) {
    mchunkptr q = (mchunkptr)old_top;
    …
    insert_chunk(m, q, psize);
  }
…
}
```

13.1.2　内存回收

在分析完内存分配之后，内存回收就易于理解了。内存释放算法最大可能地合并出连续内存块，在 top 块以后的空闲内存过大时进行 trim 操作，但是该算法存在一个不足是：

在 top 块之前如果出现了连续的空闲内存，即使这片连续的物理内存不被使用，仍然占用物理内存。因为这片内存是使用过匿名内存，在没有 swap 机制的手持设备上即不被换出，又无法释放。

```
//内存块释放函数
void mspace_free(mspace msp, void* mem) {
  if (mem != 0) {
    …
    if (!PREACTION(fm)) {
      check_inuse_chunk(fm, p);
      if (RTCHECK(ok_address(fm, p) && ok_cinuse(p))) {
        size_t psize = chunksize(p);
        mchunkptr next = chunk_plus_offset(p, psize);
```

```
/*检查前面的内存块是否处在 free 状态*/
if (!pinuse(p)) {
  /*前面的内存块 free，则找出前面内存块的大小*/
  size_t prevsize = p->prev_foot;
  if ((prevsize & IS_MMAPPED_BIT) != 0) {
    /*前面内存块是直接 map 出来的，合并 ummap 掉*/
    prevsize &= ~IS_MMAPPED_BIT;
    psize += prevsize + MMAP_FOOT_PAD;
    if (CALL_MUNMAP((char*)p - prevsize, psize) == 0)
      fm->footprint -= psize;
    goto postaction;
  }
  else {
      //算出前面内存块的指针 prev
    mchunkptr prev = chunk_minus_offset(p, prevsize);
    //两块内存块可以合并，尺寸相加
    psize += prevsize;
    p = prev;
    …
    }
    …
  }
}

if (RTCHECK(ok_next(p, next) && ok_pinuse(next))) {
  //检查下一个内存块的状态是否 free
  if (!cinuse(next)) {  /*consolidate forward*/
    //下一个内存块处于 free 状态，可以向下合并
    if (next == fm->top) {
      /*下一个内存块是本 mspace 的 top 块，这里是自由世界的边界。将本内存块释放
      到自由内存世界里去*/
      size_t tsize = fm->topsize += psize;
      fm->top = p;
      p->head = tsize | PINUSE_BIT;
      if (p == fm->dv) {
        fm->dv = 0;
        fm->dvsize = 0;
      }
      /*本 mspace 是否占用了过多的虚拟内存，如果超过边界，释放掉过多的虚拟内存*/
      if (should_trim(fm, tsize))
        sys_trim(fm, 0);
      goto postaction;
    }
    else if (next == fm->dv) {
      //next 内存块是 dv 内存块，好吧，把本块内存也合并上去
```

```
                size_t dsize = fm->dvsize += psize;
                fm->dv = p;
                set_size_and_pinuse_of_free_chunk(p, dsize);
                goto postaction;
              }
              else {
                //普通的向前内存块合并操作
                size_t nsize = chunksize(next);
                psize += nsize;
                unlink_chunk(fm, next, nsize);
                …
              }
            }
            Else
              /*无法向前合并内存，也得清除 next 内存块的 PINUSE_BIT 标志*/
              set_free_with_pinuse(p, psize, next);

            //把本块内存挂到 smallbin 链表或者 tree 里
            insert_chunk(fm, p, psize);
            check_free_chunk(fm, p);
            goto postaction;
          }
        }
      …
      }
    }
  }
```

13.2　Ashmem

匿名内存，在 PC 和服务器环境下，内核可以使用 SWAP 机制将不频繁使用的内存页面释放掉。最早期的 Android 系统也尝试采用这种方式来节省内存空间，其做法是为每个 Dalvik 进程创建一个临时的位于非易失存储介质的物理文件，将 Dalvik 的匿名虚拟内存空间 MAP 到这个文件上。这种做法与 SWAP 的机理完全相同，即利用内核的换入换出机制节省空间了。

但是随着 Android 演进，Ashmem 替换掉了原有做法。尽管作为既有内存机制的包装和利用，Ashmem 不是一个架构性的革新，但是 Ashmem 有两个值得称道的地方，一是更方便的进程间共享内存，二是更无缝的控制虚拟匿名内存对应的物理内存页面的保留和释放。

在研究 Ashmem 之前，先回顾一下 ramfs，在 rootfs 分析中遇到过 ramfs，那里 ramfs 作为 rootfs 内容的实体出现过。在实现形式上，ramfs 是一个真正的文件系统，其不同之处是它没有非易失存储介质，所有的内容都位于其节点的 Page Cache 中。Ashmem 机制下，

每个进程的匿名虚拟内存都有着一个 ramfs 的节点。

```
//在用户态使用虚拟匿名内存之前要做 MMAP 操作
static int ashmem_mmap(struct file *file, struct vm_area_struct *vma)
{
    …
    if (!asma->file) {
        /*基于传统共享内存机制，为该虚拟匿名内存创建 ramfs 节点*/
        vmfile = shmem_file_setup(name, asma->size, vma->vm_flags);
        …
        }
        …
}
/*Ashmem 基于共享内存机制实现，而无论是否配置了 tmpfs，共享内存又是基于 ramfs 来实现*/
struct file *shmem_file_setup(const char *name, loff_t size, unsigned long
flags)
{
    …
    //在 ramfs 创建一个节点
    inode = shmem_get_inode(root->d_sb, NULL, S_IFREG | S_IRWXUGO, 0, flags);
    …
    //为该节点其生产 file
    file = alloc_file(&path, FMODE_WRITE | FMODE_READ, &shmem_file_
    operations);
    …
}
```

有了这层 MAP 铺垫之后，虚拟匿名内存发生的页异常都被 int shmem_getpage(…)捕获掉，接下来就是内核中典型的 page cache 申请操作。再接下来，又是 Linux 虚拟内存的基本运行机制的老故事了，雨打风吹，换入换出，周而复始。

但是，作为基于 page cache 的文件系统，ramfs 是没法换入换出的，如果匿名内存的物理页不过就是待在那里，这跟普通的非 SWAP 机制的匿名虚拟内存有什么区别呢？

回到 Malloc，可以看到，应用运行时对内存的使用是不停地分配、释放，这当中会频繁出现整页的空闲虚拟内存。而这时候，内核并不知道这个页面是可以释放掉的。因为是匿名内存，内核只能在其不活跃时将其换出，但是这个页面并不一定处在 LRU 的队尾。所以这个时候就需要用户态通知内核将这个页面释放掉，而 Android 的确是这样做的。所以，基于 ramfs，这个动作就显得非常自然，将其 page cache 对应页面 truncate 即可。

```
/*内核记录下来所有的空闲无用内存段，待内核发现内存紧张，需要 shrink 内存，或者在 Android
下发 ASHMEM_PURGE_ALL_CACHES 命令时，删去无用页面*/
static int ashmem_shrink(struct shrinker *s, struct shrink_control *sc)
{

    list_for_each_entry_safe(range, next, &ashmem_lru_list, lru) {
        …
```

```
    /*内核中典型的 page cache 清洗函数，将删除 inode 节点对应在 start 到 end 范围内
    的所有 page  cahe 页面*/
      vmtruncate_range(inode, start, end);
      …
  }
  …
}
```

上文的源码分析里略去了 Ashmem 共享映射相关的代码，而 Ashmem 是借助传统的共享内存机制和 Binder FD 来实现其内存共享机制。Ashmem 只是其上的一层壳，这里不必再重复讨论，相关机制参见虚拟内存和 Binder 章节。

13.3　GC

尽管涉及线程与虚拟机对象管理机制等方面的问题，但是这些问题属于 Dalvik 的其他子系统，GC 本质还是 Dalvik 内存管理问题。对于 Dalvik GC 而言基本分配回收算法已被 bionic 实现，而且也许为了规避专利陷阱，Dalvik 的 GC 机制只实现了基本的 marksweep 策略，所以 Mark 机制理解清楚了，Dalvik GC 的架构就清晰了。

13.3.1　对象 Mark

Mark 试图将被引用中对象标记出来，在引用链中遍历到一个对象后，需要区分该对象的类型，下面展开分析基本的普通对象。

```
//对象遍历的基本函数
static void scanObject(const Object *obj, GcMarkContext *ctx)
{   …
    if (obj->clazz == gDvm.classJavaLangClass) {
    /*类对象的遍历，除了数据域还要遍历其静态域，这是类对象特有的情况*/
        scanClassObject(obj, ctx);
    } else if (IS_CLASS_FLAG_SET(obj->clazz, CLASS_ISARRAY)) {
    //若是数组对象，进一步对数组里的每一个对象执行 scanArrayObject
        scanArrayObject(obj, ctx);
    } else {
    //普通对象，这是展开分析的情况
        scanDataObject(obj, ctx);
    }
}

/*普通对象的遍历，需要将该对象所占用的内存标记为占用，而且要沿着该对象数据域再深一步地
遍历该对象所引用的下一级对象*/
static void scanDataObject(const Object *obj, GcMarkContext *ctx)
```

```
{   …
    //将该对象对应的位图标记为占用
    markObject((const Object *)obj->clazz, ctx);
    /*继续沿着该对象的数据域扫描，将该对象使用的对象标记为占用*/
    scanFields(obj, ctx);
    …
}
```

13.3.2　从 Root 对象集到普通对象

GC 从 root 对象开始遍历引用链，root 对象包括加载的类（也作为普通的对象存在于 Dalvik）、原始对象、全局引用及间接引用表。而与大量 Java 应用相关对象的遍历是顺着 thread 的遍历进入的，从而导致大量生成对象及其引用对象的扫描。

```
//Root 对象集扫描入口
void dvmVisitRoots(RootVisitor *visitor, void *arg)
{   …
     //已加载的类
     visitHashTable(visitor, gDvm.loadedClasses, ROOT_STICKY_CLASS, arg);
     //原始对象类
     visitPrimitiveTypes(visitor, arg);
     …
     //全局引用及间接引用表
     visitIndirectRefTable(visitor, &gDvm.jniGlobalRefTable, 0, ROOT_JNI_
     GLOBAL, arg);
     …
     visitReferenceTable(visitor, &gDvm.jniPinRefTable, 0, ROOT_VM_
     INTERNAL, arg);
     …
     //线程的遍历
     visitThreads(visitor, arg);
     …
}

//遍历线程，这里最值得关注的是对线程 stack 的遍历
static void visitThread(RootVisitor *visitor, Thread *thread, void *arg)
{
     …
     threadId = thread->threadId;
     //遍历线程对象本身
     (*visitor)(&thread->threadObj, threadId, ROOT_THREAD_OBJECT, arg);
     //遍历线程异常对象
     (*visitor)(&thread->exception, threadId, ROOT_NATIVE_STACK, arg);
     …
     //遍历 java 栈
```

```
    visitThreadStack(visitor, thread, arg);
}
```

Java 栈的遍历，将遍历层次递进到一个 Java 线程运行时产生的对象引用层面上。其中 Dalvik 栈的结构参见本书相关章节。

```
//遍历 Java 栈
static void visitThreadStack(RootVisitor *visitor, Thread *thread, void
*arg)
{   …
    //逐帧处理
    for (u4 *fp = (u4 *)thread->interpSave.curFrame;
         fp != NULL;
         fp = (u4 *)saveArea->prevFrame) {
        Method *method;
        saveArea = SAVEAREA_FROM_FP(fp);
        method = (Method *)saveArea->method;
        //检查该帧函数所占用的寄存器
        if (method != NULL && !dvmIsNativeMethod(method)) {
            …
            if (regVector == NULL) {
                …
                /*没有获得 regVector，则逐个检查该函数寄存器覆盖的内存，若为对象则执
                行遍历操作*/
                for (size_t i = 0; i < method->registersSize; ++i) {
                    //检查是否为有效对象
                    if (dvmIsValidObject((Object *)fp[i])) {
                        //遍历
                        (*visitor)(&fp[i], threadId, ROOT_JAVA_FRAME, arg);
                    }
                }
            } else {
                …
                u2 bits = 1 << 1;
                //可以直接定位对象位置，则取出对象，扫描之
                for (size_t i = 0; i < method->registersSize; ++i) {
                    …
                    if ((bits & 0x1) != 0) {
                       //遍历
                        (*visitor)(&fp[i], threadId, ROOT_JAVA_FRAME, arg);
                    }
                }
                …
            }
        }
```

```
        ...
    }
}
```

13.3.3 GC 与线程实时性

GC 是实时性的克星，是 Java 进入实时领域致命性的障碍。原因在于在 GC 线程活跃时首先要将该 Java 进程的所有线程 suspend 掉再进行 GC，否则 GC 的结论是错误的。

在 GC 线程侧，首先调用 void dvmSuspendAllThreads 通知其他 Java 线程进入 suspend 状态，其他 Java 线程的解释器在每条指令处理完毕都要调用 void dvmCheckBefore 检查 union InterpBreak 的 uint16_t subMode;且是否需要 suspend，若条件成立，则调用 dvmCheckSuspendPending 进入 suspend 状态。

```
/*GC 线程侧调用该函数停掉 Java 进程*/
void dvmSuspendAllThreads(SuspendCause why)
{
...
    //扫描 DVM 进程中所有的 Java 线程
    for (thread = gDvm.threadList; thread != NULL; thread = thread->next) {
        if (thread == self)
            continue;
        ...
        /*将每个 thread 的 int suspendCount 加一，且设置 union InterpBreak 的
        uint16_t subMode;状态，该 thread 将以 int suspendCount 和 uint16_t subMode;
        状态作为是否需要 suspend 的条件*/
        dvmAddToSuspendCounts(thread, 1,
                              (why == SUSPEND_FOR_DEBUG ||
                              why == SUSPEND_FOR_DEBUG_EVENT)
                              ? 1 : 0);
    }
    ...
    for (thread = gDvm.threadList; thread != NULL; thread = thread->next) {
        ...
        /*等待该 thread 休眠，若等待时间超过一定时限，则提升该线程的优先级，若该线程
        处于运行态只不过没有被调度上 CPU，该动作有效*/
        waitForThreadSuspend(self, thread);
       ...
    }
    ...
}

//Java 线程侧在每条指令结束之后调用该函数
void dvmCheckBefore(const u2 *pc, u4 *fp, Thread* self)
{
```

```
    …
    /*对于 GC 情况，suspendCount 为真*/
    if (self->suspendCount ||
        (self->interpBreak.ctl.subMode & kSubModeCallbackPending)) {
        …
        /*这里检查当前结束的指令属性是否为 kInstrCanBranch | kInstrCanSwitch |
        kInstrCanThrow | kInstrCanReturn 中的一种，即 Java 线程在跳转返回且不会产
        生异常时才接受 GC 的 Suspend 请求*/
        if (flags & (VERIFY_GC_INST_MASK & ~kInstrCanThrow)) {
            …
            // 检查 GC 是否活跃
            if (self->suspendCount) {
                dvmExportPC(pc, fp);
                //进入 suspend
                dvmCheckSuspendPending(self);
            }
        }
}
…
}
```

由此可以总结 GC 对实时性的影响主要是导致线程不可控、不可预测的 Suspend。在 GC 对 Java 线程 Suspend 的处理分为以下几种情况。

（1）处于运行态且获得处理器的线程，这种情况只要遇到跳转返回指令即可。

（2）处于运行态但没有获得处理器的线程，这种情况提升其优先级将有效解决问题。

（3）处于非运行态 Java 线程，这种线程在重新恢复运行时需要检查 int suspendCount 和 uint16_t subMode;以等待 GC 完成。

另外 GC 自身的效率也是影响实时性的关键因素，因为 GC 的效率决定了线程 Suspend 时间的长度。

第 14 章　进程与线程

14.1　Dalvik 虚拟机的进程

在 Dalvik 新进程创建的最关键一步是使用 Linux 的 Fork 机制从 zygote 母体 Fork 出一个新的进程来。到了这里有如下值得关注的地方。

（1）由于是 Linux 的 Fork 机制，新进程复制 Zygote 的可共享虚拟地址空间的页表页目录。而不可共享区域由 Linux 自身的 COW 机制在写时机创建。

（2）Zygote 已经进行大量的初始化，加载了大量的常用类库和二进制链接库，都被新进程继承下来。

（3）新进程如果有什么异常状况，可以轻易被 kernel 杀掉，只要 systemserver 跟 Zygote 没有问题，系统仍然可以健康的运行。

（4）在新进程运行应用代码之前构建 Android 进程安全框架。除了传统的通过用户 ID、组 ID 将进程文件访问限制在自己局部区域，通过 Capabilities 能力位更加细分地控制对内核访问。

```
//Zygote Fork 新进程的实现函数
static pid_t forkAndSpecializeCommon(const u4* args, bool isSystemServer)
{
    pid_t pid;

    //新进程的 UID 和 GID，这是在由 PackagemangeService 传送过的
    uid_t uid = (uid_t) args[0];
    …
    //调用 Linux 系统调用 Fork
    pid = fork();
    //完成 Fork 动作，新老进程都从这里返回
    if (pid == 0) {
```

至此新进程的实体被创建完毕，但是这离 Android 进程还有几步要走。首先这是由 Zygote Fork 出的进程，它具有与 Zygote 相同的权限，这是不被允许的，不然 Android 上的恶意病毒就可以为所欲为了。而新进程到了这里还没有加载 APK 的.so 和 dex，所以新进程在这里自废武功限制自己以后的所为，等到加载了恶意的 APK 也不怕伤及筋骨了。

新版 Android 的安全性更进一步，支持基于 Selinux 的面向企业应用的更高级别的安全性，但是本节摘自较早笔记，其版本没有涉及 Selinux 的适配。为了避免主题分散，针对

SEAndroid/SElinux 的分析不在本书讨论的范围之内。

```
#ifdef HAVE_ANDROID_OS
        …
        if (uid != 0) {
         …
#endif /*HAVE_ANDROID_OS*/
        //重置用户组
        err = setgroupsIntarray(gids);
        …
       //设置资源限制
        err = setrlimitsFromArray(rlimits);
        …
        //重置GID
        err = setgid(gid);
        …
        //重置UID
        err = setuid(uid);
        …
        //Capabilities位的设置
        err = setCapabilities(permittedCapabilities, effectiveCapabilities);
        …
        //设置DVM控制结构的线程号
        Thread* thread = dvmThreadSelf();
        thread->systemTid = dvmGetSysThreadId();
        …
    } else if (pid > 0) {
        /*the parent process*/
    }
    return pid;
}
```

再往后，父子进程都退到 Zygote 中，但是父进程继续等待下一个新的 DVM 进程创建申请，子进程在清空原有的 Java 栈之后开始了新生。

14.2　Dalvik 线程创建机制

Bionic 的线程机制是 Dalvik 线程的机制的基础。其实现是通过 Linux 的 Fork 机制来实现的，这样 Android 的 Java 线程实际上就是不择不扣的普通 Linux 线程，其调度完全受内核控制。

线程的运行轨迹是栈，栈是由帧组成，Dalvik 线程也不例外，首先分析 Android 代码注释里已经给出了的 bionic 线程的栈结构。

```
 * +------------------------+
 * |    pthread_internal_t  |
 * +------------------------+
 * |                        |
 * |        TLS area        |
 * |                        |
 * +------------------------+
 * |                        |
 * .                        .
 * .       stack area       .
 * .                        .
 * |                        |
 * +------------------------+
 * |       guard page       |
 * +------------------------+
```

由此可见，线程栈结构是最低层存放该线程的管理结构 pthread_internal_t；接着是线程局部存储区域；再接着是运行时产生的堆栈数据。

线程的创建实现遵循 pthread API。实现如下：

```
int pthread_create(pthread_t *thread_out, pthread_attr_t const * attr,
                   void *(*start_routine)(void *), void * arg)
{   …
    //若未准备堆栈，则分配堆栈，栈是线程的人生轨迹
    if (!attr->stack_base) {
        stack = mkstack(stackSize, attr->guard_size);
        …
    } else {
        stack = attr->stack_base;
    }
    //预留线程局部存储
    tls = (void**)(stack + stackSize - BIONIC_TLS_SLOTS*sizeof(void*));
   …
   /*调用__pthread_clone，传递下去的最重要参数是 CLONE_VM，意味着虚拟地址空间共享，
   即线程。__pthread_clone 由汇编实现，是 Bionic 的一部分，见下文分析*/
    tid =__pthread_clone((int(*)(void*))start_routine, tls,
                CLONE_FILES | CLONE_FS | CLONE_VM | CLONE_SIGHAND
                | CLONE_THREAD | CLONE_SYSVSEM | CLONE_DETACHED,
                arg);
   …
}
//__pthread_clone 的实现
ENTRY(__pthread_clone)
    .cantunwind
    /*把新线程入口地址填入堆栈，在新线程从内核退出时会从这里找到其实执行地址。其中 r1
```

```
    是堆栈地址*/
str     r0, [r1, #-4]
//系统调用需要的参数信息
    str     r3, [r1, #-8]
   // CLONE_VM 等信息放在 r0，遵循内核调用规范
    mov     r0, r2
#if __ARM_EABI__
    //保存下来 r4-r7
    stmfd   sp!, {r4, r7}
    //系统调用 clone，clone 即 Fork 的一种包装
    ldr     r7, =__NR_clone
    swi     #0
#else
    swi     #__NR_clone
#endif
    //新线程的 r0 为 0，参见内核部分
    movs    r0, r0
    …
    blt     __error
    //创建线程从这里返回
    bxne    lr
    //新线程继续执行
    //取出存放在新线程堆栈里起始地址和参数
    ldr     r0, [sp, #-4]
    ldr     r1, [sp, #-8]
    //新线程的堆栈指针指向 TLS 区域，参见上文堆栈结构
    mov     r2, sp              @ __thread_entry needs the TLS pointer
    sub     lr, lr
    //新线程继续调用__thread_entry
    b       __thread_entry
    …
END(__pthread_clone)

//新线程的 trampoline
void __thread_entry(int (*func)(void*), void *arg, void **tls)
{
    int retValue;
    pthread_internal_t * thrInfo;
    …
    thrInfo = (pthread_internal_t *) tls[TLS_SLOT_THREAD_ID];
   //初始化新线程的线程局部存储
    __init_tls( tls, thrInfo );
    //跳转到新线程的起始地址
    pthread_exit( (void*)func(arg) );
}
```

14.3 Android 线程模型

Android 进程天生是多进程的。Android 的线程结构有如下几种。

（1）主线程，由 Zygote 母体生成。

（2）线程池的线程在用户态二进制层创建，但是同时创建 DVM 需要的 Context，这些线程爬在 Binder 上，实现 Binder 协议，受 Binder 事件驱动往上蹿到 Java 层完成跨进程调用（有时也是进程内调用）。

（3）Java 应用创建的线程，普通的 Dalvik 线程。

（4）在用户态二进制层创建的 pthread 线程没有相应的 DVM Contex，无法蹿到 Java 层，只能执行二进制机器码，通常做一些 deamon 工作。

前两者线程是 Android 应用框架的默认生成，后两种线程由应用、中间件生成。本节着重分析前两种线程的生成。

14.3.1 主线程的生成

在 Zygote 母体 Fork 出 Android 应用之后，主线程作为新的 Android 应用的第一个线程就存在了。

```
    public static final void zygoteInit(int targetSdkVersion, String[] argv)
            throws ZygoteInit.MethodAndArgsCaller {
        ...
/*从这里创建本进程的线程池*/
        zygoteInitNative();
/*主线程继续从这里演化，activitythread 加载等工作在这里进行*/
        applicationInit(targetSdkVersion, argv);
    }
```

14.3.2 线程池线程的生成

主线程实体在 Java 层调用 class RuntimeInit 的 public static final native void zygoteInitNative();。

C++层的 class AppRuntime 的 virtual void onZygoteInit()函数完成实际的工作线程创建工作，线程池的创建步骤是，首先创建第一个线程 A，然后线程 A 趴在 Binder 上监听 BR_SPAWN_LOOPER 事件，该事件发生时，线程 A 创建线程 B，线程 B 继续监听 BR_SPAWN_LOOPER 事件，该事件发生时，线程 B 创建线程 C。所以线程池线程一共会生成 3 个。这是 Binder 协议决定的，根据系统处理器数目以及应用进程的负载强度，线程池线程的数目可以动态调整，当然这属于 Android 优化的考虑，不是 Google Android Release 的实现。

```
virtual void onZygoteInit()
    {
        sp<ProcessState> proc = ProcessState::self();
        /*创建线程池里的第一个线程*/
        proc->startThreadPool();
}
```

线程池的创建函数与普通的非 Java 线程创建区别的是，这将创建具有能够进入 DVM 运行时能力的线程。

```
int AndroidRuntime::javaCreateThreadEtc(…)
{   …
       /*entryFunction 即为 class PoolThread :: virtual bool threadLoop()*/
       args[0] = (void*) entryFunction;
   …
/*AndroidRuntime::javaThreadShell 是关键的包裹函数，它为新线程创建出需要的 DVM
context，且调用 entryFunction 运行*/
    result = androidCreateRawThreadEtc(AndroidRuntime::javaThreadShell, args,
        threadName, threadPriority, threadStackSize, threadId);
   …
}
int AndroidRuntime::javaThreadShell(void* args) {
  …
    /*为本线程创建 Java 运行所需的 context，该部分工作与 Dalvik 线程创建一致，具体分
    析见 Dalvik 线程部分*/
    if (javaAttachThread(name, &env) != JNI_OK)
        return -1;
    /*环境搭好了，干活 class PoolThread :: virtual bool threadLoop()*/
    result = (*(android_thread_func_t)start)(userData);
…
}
```

线程池里的每一线程都继承自 class PoolThread，threadLoop()是其主要工作。

```
class PoolThread : public Thread
{…
protected:
/*线程池线程的主要工作，即 Binder 协议的执行。本节分析线程池线程的线程结构，其工作内容
部分的分析见 Binder 部分*/
    virtual bool threadLoop()
    {
        IPCThreadState::self()->joinThreadPool(mIsMain);
        return false;
    }
    …
};
```

14.4　Java 线程转换

本节分析线程从 Java 代码调用 JNI 函数以及具有 Java 执行能力的线程从 JNI 代码调用 Java 函数的过程。

14.4.1　从 Java 到 JNI

从 Java 到 JNI 即解释器遇到了 native 函数调用。这是从解释器向二进制运行时转向的动作，发生在当解释器遇到函数调用指令，且发现欲调用的函数是 native 类型。接下来以 C 解释器为例分析 native 函数调用，关于寄存器 Context 的处理在“Dalvik 寄存器编译模型”一节中进行分析，本节着重与分析 native 调用相关。

```
//解释器对 invokeMethod 的 handler
GOTO_TARGET(invokeMethod,    bool    methodCallRange,    const    Method*
_methodToCall,
    u2 count, u2 regs)
    {   …
        newSaveArea->method = methodToCall;
        if (self->interpBreak.ctl.subMode != 0) {
            /*GC 或者调试发生，记下指令位置，准备在该当前指令完成后退出解释器*/
            PC_TO_SELF();
            dvmReportInvoke(self, methodToCall);
        }
        if (!dvmIsNativeMethod(methodToCall)) {
         //Java 函数的调用，不在本节分析
          …
        } else {
            …
            self->interpSave.curFrame = newFp;
            …
            /*调用 native 函数,newFp 作为参数列表指针传递给 jni native 函数。这里 Java
            栈将不再变化，转而进入二进制线程栈。而实际上在类加载是 native 函数，被初始
            化为 void dvmResolveNativeMethod(…)，这类似一个弹簧的功能，接下来从
            这里再跳到真正的 native 函数中*/
            (*methodToCall->nativeFunc)(newFp, &retval, methodToCall, self);
            …
            /*函数调用完毕，退一帧*/
            dvmPopJniLocals(self, newSaveArea);
            self->interpSave.curFrame = newSaveArea->prevFrame;
            fp = newSaveArea->prevFrame;
            …
```

```
            }
            …
GOTO_TARGET_END

//native 函数跳转中间函数，这里将解析 native 函数真正地址并跳转
void dvmResolveNativeMethod(const u4* args, JValue* pResult,
    const Method* method, Thread* self)
{
    …
    //在缓冲机制中查找该 native 函数
    DalvikNativeFunc infunc = dvmLookupInternalNativeMethod(method);
    if (infunc != NULL) {
        …
        //该函数已被使用过
        DalvikBridgeFunc dfunc = (DalvikBridgeFunc) infunc;
        //写入 method->nativeFunc，下一次不用弹簧函数了
        dvmSetNativeFunc((Method*) method, dfunc, NULL);
        //调用 native 函数
        dfunc(args, pResult, method, self);
        return;
    }

    /*在.so 库文件里查找该 native 函数，本书没有关于.so 文件查找函数的分析，但可以参考
    linker 一节，工作原理类似*/
    void* func = lookupSharedLibMethod(method);
    if (func != NULL) {
        /*放入缓冲，写入 method->nativeFunc*/
        dvmUseJNIBridge((Method*) method, func);
        //native 函数调用
        (*method->nativeFunc)(args, pResult, method, self);
        return;
    }
   …
}
```

14.4.2　从 JNI 到 Java

JNI 调用 Java 函数，本质上通过解释器运行 Java 字节码，类似于解释器遇到 invoke 指令。但是解释器处理 invoke 函数之前有一个运行时的 Context，包括调用参数寄存器分配、函数局部变量寄存器分配、函数返回值寄存器分配，以及 DVM 当前状态。除了后者是现成的，其余 Context 都需要动态搭建起来。为了完成这些工作，DVM 为此准备了一个函数表，向 JNI 提供不同类型的 Java 函数调用。

```
//DVM 向 JNI 提供服务的函数表
```

```
    static const struct JNINativeInterface gNativeInterface = {
    …
    CallByteMethod,
    CallByteMethodV,
    CallByteMethodA,
    …
}
```

在 JNI 层针对不同类型的 Java 函数选择其不同的入口函数。这些入口函数尽管与不同类型的 Java 函数有着不同的实现，但是其实现方式主体基本相同，即借助 void dvmCallMethodX(…)实现调用。

以 void dvmCallMethodV(…)为例，代码如下：

```
/*这之前要找到当前代码所在线程 Thread* self、要调用的函数 Method* method、若不是
static 函数还要找出其对象 Object* obj/
void dvmCallMethodV(Thread* self, const Method* method, Object* obj,
    bool fromJni, JValue* pResult, va_list args)
{
    …
    //
    clazz = callPrep(self, method, obj, false);
    if (clazz == NULL)
        return;

    /*"ins" for new frame start at frame pointer plus locals*/
    ins = ((u4*)self->interpSave.curFrame) +
           (method->registersSize - method->insSize);

   //非静态函数的调用要将对象放入 context 中
    if (!dvmIsStaticMethod(method)) {
        …
        *ins++ = (u4) obj;
        verifyCount++;
    }
   //根据参数定义将参数填入栈中
    while (*desc != '\0') {
            …
            case 'F': {
                …
                 //浮点类型参数
                *ins++ = dvmFloatToU4(f);
                …
                break;
            }
            …
```

```
            default: {
                /*简单类型参数直接复制*/
                *ins++ = va_arg(args, u4);
                …
                break;
            }
        }
    }
    …
    //进入解释器
    dvmInterpret(self, method, pResult);

    //java 调用完毕弹掉 java 栈
    dvmPopFrame(self);
}
```

DVM Java 函数看到的是 DVM 虚拟机，其字节码直接操作虚拟机 Context，没有 Context 其字节码是没有意义的，所以 Context 的准备是执行 DVM Java 函数的前提。

```
static ClassObject* callPrep(Thread* self, const Method* method, Object* obj,
    bool checkAccess)
{
    ClassObject* clazz;

    …
    //找到对应类
    if (obj != NULL)
        clazz = obj->clazz;
    else
        clazz = method->clazz;
    …

if (dvmIsNativeMethod(method)) {
        //native 函数的调用，这里不考虑
        …
    } else {
        /*虚拟机的 Context 都在其 Java 栈中存放，一个函数调用 Java 栈增长一帧，这里根
        据调用函数的寄存器使用个数、参数、局部变量所占用的空间分配栈空间。其格式参见
        "Dalvik 寄存器模型"一节*/
        if (!dvmPushInterpFrame(self, method)) {
            …
            return NULL;
```

```
            }
        }

        return clazz;
    }
```

在将 Java 函数的 Context 准备好之后进入解释器，尽管解释器物理上仍然与当前二进制码使用同一个栈，但逻辑上，当前线程的栈已经切换到了 Java 栈。

第 15 章　Bionic 的动态加载机制

一个 OS 的基础组件中，最关键、使用最频繁的就是 C 库，C 库是系统其他各组件工作的基础。5 年前移动和嵌入式领域的主流处理器还是 ARM9，而 Linux 软件体系使用的 C 库是 GLIBC。GLIBC 相对较大，编译完之后的体积足有 8M 多。大家知道应用对 C 库的访问是必须和频繁的，而一个大型的 C 库面临的问题是更松散的系统调用分布，将导致更频繁缺页异常，对于当时的 ARM9 处理器 GLIBC 苦不堪言。程序的启动时间很大程度消耗在这个庞大 C 库上。于是出现了裁剪过的 ucLibc，但是出现了兼容性问题，而且仍旧过大。

最初的 Android 设计目标是 400M 的 ARM9，而且由于 Android 不需要广泛的支持大范围 Linux 二进制程序，C 库兼容性的必要性降低了。所以 Android 重新设计 C 库，这是个精巧的 C 库，避免使用 GLIBC 和 ucLibc 而产生的庞大开销。最初的 Android 版本能够在 ARM9 上顺利运行，这个小巧的 C 库功不可没。

现在嵌入式和移动处理器的性能越来越接近 PC，一颗 4 核 CA9 的处理器足以运行 GLIBC 了，而小巧高效 Bionic 一直没替换掉，其原因也许在于由于 Android 不广泛支持二进制应用，所以也不需要一个大型的 C 库吧。

Bionic C 库的实现诸如 malloc 机制、线程机制等与系统其他部分紧密相关，所以都被分散到了其他章节，Bionic C 库机制仅剩下动态加载机制单独作为本章的内容。

不同于传统 C 库的 ld.so，Bionic C 库重新实现 Linker 来完成其功能。Linker 的作用是将依靠动态链接库二进制可执行程序的动态链接库依次找出来逐个解析链接，然后再跳到二进制可执行程序的入口。这其中对动态库的链接是个递归的过程，因为一个动态库可以本身也依赖于另一个动态库，Linker 需要完成这个递归加载的过程。所以 Linker 的分析也需要采用这个逐级递归逻辑。

15.1　Linker——用户态入口

内核运行新的基于动态加载应用时，首先要找到其对应的加载器，并将其映射。在内核态切换到用户态时，执行的第一条指令并不在应用的二进制镜像里起始地址，而是动态加载器的始地址，代码如下：

```
_start://用户态的第一条指令
    mov r0, sp  //内核给攒用户栈，里面放着启动参数
    mov r1, #0
    bl  __linker_init  //Linker 的主体，这里完成动态库的逐级解析、加载工作
```

```
/* linker init returns the _entry address in the main image */
/*Linker 会找到二进制应用入口点，并将其放在 r0，下一步就是跳到二进制应用入口点*/
mov pc, r0
```

15.2 Linker 主体——link_image

Linker 的主体是一级级对 ELF 文件解析其依赖的动态库并链接。

（1）第一级：加载需要的动态库，并将该 ELF 镜像链接上去。

```
//参数 soinfo *si 是对这个需要链接动态库的 ELF 文件的描述
static int link_image(soinfo *si, unsigned wr_offset)
{
    ...

    /* if this is the main executable, then load all of the preloads now */
    // LD_PRELOAD 的处理，这里不讨论
    if(si->flags & FLAG_EXE) {
        int i;
        memset(preloads, 0, sizeof(preloads));
        for(i = 0; ldpreload_names[i] != NULL; i++) {
            soinfo *lsi = find_library(ldpreload_names[i]);

        }
    }
    //依次从 ELF 里找出需要动态加载动态库，并逐一加载

    for(d = si->dynamic; *d; d += 2) {
        /*si->strtab + d[1]就是动态库的文件名，以/system/bin/app_process 为例，
        这里会解析出所依赖的动态库如下：liblog.so、libbinder.so、libandroid_
        runtime.so、libc.so、libstdc++.so、libm.so。Linker 会去默认的/vendor/
        lib、/system/lib 两个目录下寻找这些库*/
        if(d[0] == DT_NEEDED){
        //寻找需要的动态库，并加载之
            soinfo *lsi = find_library(si->strtab + d[1]);

        }
    }

//真正的链接在这里进行
    if(si->plt_rel) {
        if(reloc_library(si, si->plt_rel, si->plt_rel_count))
            goto fail;
    }
    if(si->rel) {
```

```
        if(reloc_library(si, si->rel, si->rel_count))
            goto fail;
    }


}
```

（2）第二级：在已加载库里检查需要的动态库时已经加载并链接，否则将加载、链接该动态库。

```
soinfo *find_library(const char *name)
{
   //已加载、链接的动态库都放在 solist 链表里
    for(si = solist; si != 0; si = si->next){
        if(!strcmp(bname, si->name)) {
            … solist
  //该库已加载并链接，直接返回即可
            if(si->flags & FLAG_LINKED) return si;
           …
        }
}
//该库已并未涉及，加载链接之
si = load_library(name);
…
  /*该动态库已经被加载，下一步要检查该动态库本身是否依赖其他动态库，如果存在依赖则递归
  进去为该动态库加载其依赖的动态库*/
  return init_library(si);
}
```

不由得啰嗦一句：这里对 solist 链表的操作没有任何互斥操作，其原因在于，这时进程是独立的，且是单线程。

```
//加载动态库所依赖的动态库
static soinfo *init_library(soinfo *si)
{
    …
    //递归加载从这里进去
       if(link_image(si, wr_offset)) {
            /* We failed to link. However, we can only restore libbase
            ** if no additional libraries have moved it since we updated it.
            */
        munmap((void *)si->base, si->size);
        return NULL;
    }

    return si;
}
```

（3）第三级：加载动态库。

```
static soinfo *
load_library(const char *name)
{
//打开该动态库文件
   int fd = open_library(name);

   //给该动态库分配管理结构并加入 solist 链表
    si = alloc_info(bname ? bname + 1 : name);

    /* Now actually load the library's segments into right places in memory
*/
    //具体对动态库文件的加载操作，把每个 PT_LOAD 类型的段 MAP 出来
    if (load_segments(fd, &__header[0], si) < 0) {
        goto fail;
    }
    …
}
```

第 16 章　Android 系统初始

16.1　Android 入口

本节从进程角度上分析如何从 Init 进程跑到 ZygoteInit.main (String argv[])。

首先作为用户态的第一个线程，Init 是被内核加载的，对于 Android 系统即为/init，参见内核部分的 rootfs。Init 进程第一件工作是执行由 init.rc 里列出的命令。根据 init.rc 里的指令，Init 进程要启动如下 service：

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --
start-system-server
    socket zygote stream 666
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
```

该 service 其实就是位于/system/bin/app_process 的进程，其入口函数源码所在文件在 frameworks/base/cmds/app_process/app_main.cpp。

```
int main(int argc, const char* const argv[])
{
    …
    /*class AndroidRuntime 实际上被每个 dvm 进程所使用，但是其初始化只在这里进行。
    每个 dvm 进程是被 fork()出来，所以可以继续使用其父进程的 Runtime 实例。而初始化只
    在这里进行一次。而 class AppRuntime 继承自 AndroidRuntime，其初始化函数里将进
    行 skia 图形相关初始化*/
    AppRuntime runtime;
    const char *arg;
    …

    // Next arg is startup classname or "--zygote"
    if (i < argc) {
        arg = argv[i++];
        /*根据 init.rc 的启动参数: --zygote --start-system-server*/
        if (0 == strcmp("--zygote", arg)) {
            //条件成立
            bool startSystemServer = (i < argc) ?
                    strcmp(argv[i], "--start-system-server") == 0 : false;
```

```
            setArgv0(argv0, "zygote");
            //app_process 就是 zygote
            set_process_name("zygote");
/*com.android.internal.os.ZygoteInit 是 Zygote 进程演化为 dvm 的入口，Zygote 还
有一个任务是启动 SystemServer*/
            runtime.start("com.android.internal.os.ZygoteInit",
               startSystemServer);
        } else {
         …
        }
    } else {
        …
    }

}

/*Android 运行时*/
void AndroidRuntime::start(const char* className, const bool startSystemServer)
{
   …
   //初始化 dvm 的运行环境
    if (startVm(&mJavaVM, &env) != 0)
        goto bail;
   …
    //找到需要执行的 class：com.android.internal.os.ZygoteInit
    startClass = env->FindClass(slashClassName);
    if (startClass == NULL) {
       …
    } else {
        //找到 com.android.internal.os.ZygoteInit 的 main 函数
        startMeth = env->GetStaticMethodID(startClass, "main",
            "([Ljava/lang/String;)V");
        if (startMeth == NULL) {
            LOGE("JavaVM unable to find main() in '%s'\n", className);
            /*keep going*/
        } else {
            /*com.android.internal.os.ZygoteInit 的 main 函数是 dvm 世界的入口*/
            env->CallStaticVoidMethod(startClass, startMeth, strArray);
         …
        }
    }…
}
```

16.2　Init——OS 的入口

Init 进程是第一个用户态进程，是操作系统的入口。Init 进程在不同的 OS 里有不同的实现，其主要工作是根据 RC 文件的指示工作，完成系统初始化。

16.2.1　RC 文件分析

RC 文件是一种脚本文件，里面记录了启动操作系统各组件的动作。针对于不同的系统配置，RC 文件有着不同的内容，但通常是将硬件基础设施构建完成之后，启动 Android 的系统服务。从结构来看 RC 文件为两级结构。

（1）第一级是 section，section 有以下三类。

① 以 on 开头的 section，这个 section 里会执行一系列命令。

② 以 service 开头的 section，这个 section 指定一个 service 及其相关参数。

③ 以 import 开头的 section。

（2）第二级是针对于每个 section 动作和参数。

Init 对 RC 分析的主要工作如下：

```
static void parse_config(const char *fn, char *s)
{   …
    for (;;) {
        switch (next_token(&state)) {
        case T_EOF:
            state.parse_line(&state, 0, 0);
            return;
        case T_NEWLINE: /*遇到了新的一行，现在处理上一行，上一行分析结果都位于 args[]
        数组中*/
            if (nargs) {
                int kw = lookup_keyword(args[0]); /*首先去看这一行打头的关键字是
                什么*/
                if (kw_is(kw, SECTION)) { /*如果一行以 on 或 service,import 打头，
                该表达式为真*/
                    state.parse_line(&state, 0, 0);
/*这里分析一级语句的类型，如果是 on，section，就用 static void parse_line_
service(…)来分析这一 section；如果是 service，section 就用 static void
parse_line_action(…)来分析这一 section*/
                    parse_new_section(&state, kw, nargs, args);
                } else {
/*分析二级语句的具体的命令和参数，二级语句的关键字和参数在这之前都被放入 args 数组中了*/
                    state.parse_line(&state, nargs, args);
                }
                nargs = 0;
```

```
            }
            break;
        case T_TEXT:
/*对于一、二级语句行，这里分析关键字及其后面的文本参数，每遇到关键字和参数都会走到这里一次，作为结果，每个关键字和参数都被放到 args[]数组里*/
            if (nargs < SVC_MAXARGS) {
                args[nargs++] = state.text;
            }
            break;
        }
    }
}


void parse_new_section(struct parse_state *state, int kw,
                       int nargs, char **args)
{   …
    switch(kw) {
    case K_service: //section 为 service
        state->context = parse_service(state, nargs, args);//分析该 service
        if (state->context) {
//对 service 的每个属性行用 static void parse_line_service (…)函数分析
            state->parse_line = parse_line_service;
            return;
        }
        break;
    case K_on: //section 为 on
        state->context = parse_action(state, nargs, args); //对 on 进行分析
        if (state->context) {
//对 on section 里的每个命令行用 static void parse_line_action(…)函数分析
            state->parse_line = parse_line_action;
            return;
        }
        break;
    }
    state->parse_line = parse_line_no_op;
    …
}
```

（3）分析 service。

Service 的基本结构如下：

```
struct service {
    /*所有的 service 穿成一个以 service_list 为首的链表*/
```

```
    struct listnode slist;
    //该 service 的名称
    const char *name;
    /*该 service 所属的 classname，init 进程会依据不同的 classname 启动与该
    classname 匹配的所有 service——void service_for_each_class(…)*/
    const char *classname;
    /该 service 的 flag 标志,init 进程会依据不同的标志启动与该标志匹配的所有 service
    ——void service_for_each_flags (…)*/
    unsigned flags;
    …
    //在 restart 时，该 service 执行的 action，由多个命令组成
    struct action onrestart;  /*Actions to execute on restart*/

    /*keycodes for triggering this service via /dev/keychord*/
    …
    }


static void *parse_service(struct parse_state *state, int nargs, char **args)
{
    …
    svc = calloc(1, sizeof(*svc) + sizeof(char*) * nargs);/*创建一个 struct
    service 结构来表示该 service*/
    …
    svc->name = args[1];//该 service 的名字
    /*classname 默认值为 default，在分析该 service 组成语句时会填充该值*/
    svc->classname = "default";
    memcpy(svc->args, args + 2, sizeof(char*) * nargs);//保存 service 的参数
    svc->args[nargs] = 0;
    svc->nargs = nargs;
    svc->onrestart.name = "onrestart";
    //初始化 onstart action 的命令行链表
    list_init(&svc->onrestart.commands);
    //将该 service 加到系统的 service 链表中
    list_add_tail(&service_list, &svc->slist);
    return svc;
}
```

对于 serice 下面的每一个属性行，都用 static void parse_line_service(…)来分析，这个函数很长，其骨架结构如下：

```
static void parse_line_service(struct parse_state *state, int nargs, char
**args)
{

    kw = lookup_keyword(args[0]);//找出关键字
```

```
switch (kw) {
…
case K_class:                      //是否显式指定 classname
    if (nargs != 2) {
        parse_error(state, "class option requires a classname\n");
    } else {
        svc->classname = args[1]; //保存指定 classname
    }
    break;
 …
case K_disabled:                   //是否默认为 disable 状态
    svc->flags |= SVC_DISABLED;    //设置禁止 flag
    break;

case K_oneshot:                    //是否为 oneshot 模式
    svc->flags |= SVC_ONESHOT;     //设置 oneshot flag
    break;
  case K_onrestart:
 /*分析某 service 对应的 onrestart 动作，以 Zygote 为例，该 service 对于多个
 onrestart 动作：
service zygote…
…
每个 onrestart 动作都对应一个命令
onrestart write /sys/android_power/request_state wake
onrestart write /sys/power/state on
onrestart restart surfaceflinger
onrestart restart media
onrestart restart netd
所有命令都搜集起来，作为 onrestart action 的命令列表
*/
    nargs--;
    args++;
    //提取命令类型
    kw = lookup_keyword(args[0]);
    …
    //分配命令结构
    cmd = malloc(sizeof(*cmd) + sizeof(char*) * nargs);
    //提取命令动作函数
    cmd->func = kw_func(kw);
    cmd->nargs = nargs;
    //存储命令参数
    memcpy(cmd->args, args, sizeof(char*) * nargs);
    //加入 onrestart action 命令链表
    list_add_tail(&svc->onrestart.commands, &cmd->clist);
    break;
```

```
    ...
}
```

（4）“on” section 的分析。

每个 “on” section 表示在某个时刻触发一系列动作，其基本结构如下：

```
struct action {
    /*所有的 action 穿成以 action_list 为首的链表*/
    struct listnode alist;
   /*以 action_queue 为首的链表记录了即将被 init 执行的 action 动作链表，qlist 是这
   个链表的节点*/
    struct listnode qlist;

   //该 action 对应的命令链表
    struct listnode commands;
   //指向当前命令
    struct command *current;
};

static void *parse_action(struct parse_state *state, int nargs, char **args)
{
    ...
    act = calloc(1, sizeof(*act));//对于每个 on,section 都生成一个 struct action
    act->name = args[1];  //这里的 args[1]就是 on 的触发条件，如 init、boot 等
    list_init(&act->commands);
    list_add_tail(&action_list, &act->alist);/*并将这个 struct action 加入系
    统的 action_list 链表*/
    ...
    return act;
}
```

对于每个 on，section 里的命令都生成一个 struct command 结构，并将这个结构加入在 struct action 的 commands 列表中。

On section 的二级语句分析如下：

```
static void parse_line_action(struct parse_state* state, int nargs, char
**args)
{
    ...
    kw = lookup_keyword(args[0]);    //找到命令对应的关键字

    cmd = malloc(sizeof(*cmd) + sizeof(char*) * nargs);
    cmd->func = kw_func(kw);          //找到每个命令对应的执行函数
    cmd->nargs = nargs;
    memcpy(cmd->args, args, sizeof(char*) * nargs);
    list_add_tail(&act->commands, &cmd->clist);
```

```
}
```

分析出来的 on section 被放入 action_list 链表中。

16.2.2 RC 动作执行

RC 定义的动作和 service 分析后都被放到 action_list 链表和 service_list 链表中。

（1）On section 动作的执行。

每个 on section 都有着自己的执行时机，on 之后的字符串指出了该 action 在合适时被执行。在 init 进程完成对 RC 文件的分析之后调用 action_for_each_trigger("XXX", action_add_queue_tail);将不同执行时机的 action 加入 action 的执行链表 action_queue。接着 init 的主循环中依次触发每个 action。Init 进程执行 action 的动作顺序如下：

"early-init"->"init"->"early-fs"->"fs"->"post-fs"->"post-fs-data"->"early-boot"->"boot"

除了在 RC 文件里定义的 action 之外，Init 还有一些 builtin 的 action，init 使用 void queue_builtin_action(…)将其现在合适的位置加入 action_queue 执行链表。

如在触发时机 early-init 与 init 之间，直接加入如下 builtin action，部分代码如下：

```
{
…
//将触发时机 early-init 的 action 加入 action_queue 执行链表
action_for_each_trigger("early-init", action_add_queue_tail);
    //如下是 init 的系统默认的 builtin action
    queue_builtin_action(wait_for_coldboot_done_action,"wait_for_
    coldboot_done");
    queue_builtin_action(property_init_action, "property_init");
    queue_builtin_action(keychord_init_action, "keychord_init");
    queue_builtin_action(console_init_action, "console_init");
    queue_builtin_action(set_init_properties_action, "set_init_properties");

    /*将触发时机"init"的 action 加入 action_queue 执行链表*/
    action_for_each_trigger("init", action_add_queue_tail);
…
}
```

（2）Service section 中 service 的执行。

① 依据 classname 启动。

对于每个 service section，每个 class 都会有一个属性 classname，android 可以启动所有 classname 符合条件的 service。常见的一个例子是 on boot section 里会默认启动，classname 为 core 和 main 的 service：

在 rc 文件中有如下定义：

```
on boot
class_start core
class_start main
```

而 class_start 命令对应的执行函数为定义如下（在 system/core/init/keywords.h 中）：

```
KEYWORD(class_start, COMMAND, 1, do_class_start)
```

int do_class_start(…)会依据 classname 启动对于的 service。

② 启动指定名字的 service。

根据 service 结构的成员变量 const char *name;来启动某个特定的 service，这种方式常用在 RC 文件里通过命令启动启动某个 service，如在 RC 文件中有如下定义：

```
on property:persist.service.adb.enable=1
start adbd
```

而 start 命令对应的执行函数为定义如下（在 system/core/init/keywords.h 中）：

```
KEYWORD(start,       COMMAND, 1, do_start)
```

int do_start(…)函数会启动名为 adbd 的 service。

③ 以某个标志位启动 service。

启动标志位为 SVC_RESTARTING 的 service。

```
static void restart_processes()
{
    process_needs_restart = 0;
    service_for_each_flags(SVC_RESTARTING,
                           restart_service_if_needed);
}
```

16.2.3　RC 的逻辑分析

RC 文件虽然只是脚本文件，但是控制着整个 Android 系统的启动，所以还是值得一读的。RC 文件分析依据的脉络是几个主要的启动时机：

early-init→init→early-fs→fs→post-fs→post-fs-data→early-boot→boot

（1）early-init。启动 ueventd，设备管理的基础，见下文分析。

（2）init。时间设置，设置环境变量，创建基本目录。

（3）fs。通常与具体的硬件平台有关，这里 mount 上/system、/data。

（4）post-fs。某些目录权限的修改。

（5）post-fs-data。/data 下某些目录的创建。

（6）boot。网络、蓝牙、alsa、ril 等设备的属性设置，启动 core 和 main 之类的 sevice。

16.2.4　设备探测

Android 利用 Linux 的 uevent 探测系统中的设备，再根据动态探测到设备信息动态创建/dev 下的设备文件，代码如下：

```
int device_init(void)
{   …
    fd = open_uevent_socket();  //首先建立与内核的联系管道:通过 NETLINK
    …
    coldboot(fd, "/sys/class"); //扫描以下目录中的设备并为其动态创建设备文件
    coldboot(fd, "/sys/block");
    coldboot(fd, "/sys/devices");
    …
}
```

这里有两个层次的动作，首先建立起与内核 uevent 机制的联系，内核 uevent 机制通过两种方式与用户态沟通：/sbin/hotplug 和 NETLINK。

/sbin/hotplug 是用户态的一个系统进程，每当内核触发一个 uevent 事件都会把事件描述当作参数，启动/sbin/hotplug，而/sbin/hotplug 启动后根据内核传来的参数分析 uevent 事件，进而与 OS 系统组件交互。Android 体系抛弃了使用/sbin/hotplug 的方式，而依赖与 NETLINK 取得 uevent 消息。

```
static void do_coldboot(int event_fd, DIR *d)
{
    …
    fd = openat(dfd, "uevent", O_WRONLY);//打开该目录下的 uevent 文件
    if(fd >= 0) {
        write(fd, "add\n", 4); //向该 uevent 文件写入 add
        close(fd);
        handle_device_fd(event_fd);/*由于激活了 uevent 文件的 add 动作，下面可以
        从 NETLINK 取 uevent 信息了*/
    }
    //接下来在该目录下寻找目录项
    while((de = readdir(d))) {
        DIR *d2;

        if(de->d_type != DT_DIR || de->d_name[0] == '.')
            continue;  //不是目录文件，绕过

        fd = openat(dfd, de->d_name, O_RDONLY | O_DIRECTORY); //新发现一个目录
        …
        d2 = fdopendir(fd);
        …
            do_coldboot(event_fd, d2);/*递归进入这个目录，再次寻找这个目录下的
            uevent 文件
        …
    }
```

这个函数的主要动作就是到一个目录下找 uevent 文件，并通过向这个文件写入 add 激活其 uevent 的信息，然后通过 void handle_device_fd(…)从 NETLINK 获得 uevent 描述，再

根据获得的设备信息动态创建/dev 下的设备文件。/dev 创建的具体执行过程如图 16-1 所示。

void handle_device_fd(…)-> static void handle_device_event(…)-> void make_device(…)

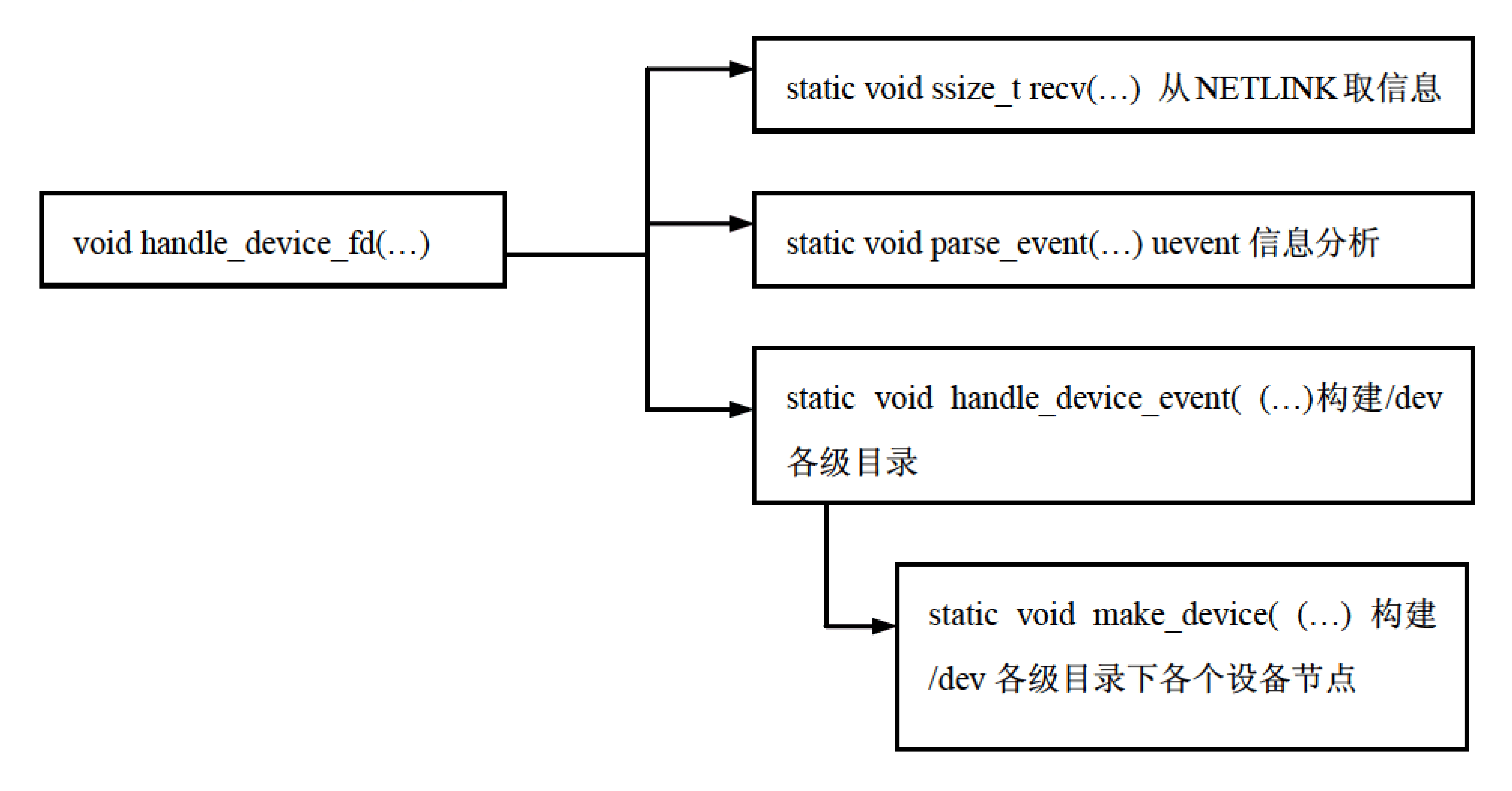


图 16-1　/dev 创建的具体执行过程

隐藏在这之下的内核机制如下。

（1）每当驱动向内核注册一个设备，那么内核为其在 sysfs 文件系统里创建一个目录。

（2）内核在该设备对应目录下放置一个 uevent 文件，而这个文件写函数，会触发 uevent。

（3）当用户态往这个设备目录下的 uevent 文件写入 add 动作时，该文件下的函数被触发，进而触发内核的 uevent 机制。

（4）内核 uevent 机制向 NETLINK 写入设备的信息。

（5）Android 用户态从 NETLINK 取出设备信息。

16.2.5　property 库的构建

Android 把系统的属性值统一存放在 property 库，Android 系统其他组件通过 int property_set(…)来设置相关属性值，而通过 int property_get(…)来获取相关属性值。

关于 property 库的构建有如下几个方面的内容。

（1）init 进程调用 void property_init(…)建立存放 property 值的内存区域，并将系统中默认的 property 配置文件/default.prop 加载到内存中。

（2）RC 文件以及 Init 进程中动态设置系统 property。

（3）Init 进程在 post-fs-data 与 early-boot 中间，调用 queue_builtin_action(property_service_init_action, "property_service_init");将 property 初始化的 builtin action 加入运行队列。该 action 将系统文件/system/build.prop、/system/default.prop、/data/local.prop 里的 property 值加载入内存。而更为重要的是，Init 进程创建一个 PF_UNIX 域的进程间通信 socket，以后 Android 其他组件（由于与 Init 进程不在同一进程空间）就通过这个 socket 机制来查询和更新 property 库。

另一个值得关注的地方是，某个 property 改变而将会导致系统触发一系列的动作，这种类型的 property 在 RC 文件被当成一个 on section 被定义，代码如下：

```
on property:vold.decrypt=trigger_restart_framework
    class_start main
    class_start late_start
```

Android 运行时，每当这种 property 值发生改变，action_list 里每一项都受到检查，如果发现该项的触发条件恰好是 property 值，那么该 section 将被触发。

```
void property_changed(const char *name, const char *value)
{
    if (property_triggers_enabled) {//init 启动后 property_triggers_enabled
    被置 1
        //检查与 property 值相关的 action
        queue_property_triggers(name, value);
         }
}
void queue_property_triggers(const char *name, const char *value)
{
    struct listnode *node;
    struct action *act;
    list_for_each(node, &action_list) {//遍历 action_list 链表
     //取出每个 action
        act = node_to_item(node, struct action, alist);
        //检查该 action 是否由 property 触发
        if (!strncmp(act->name, "property:", strlen("property:"))) {
            const char *test = act->name + strlen("property:");
            int name_length = strlen(name);
            //检查 property 的属性名
            if (!strncmp(name, test, name_length) &&
                    test[name_length] == '=' &&
                    !strcmp(test + name_length + 1, value)) {
                //加入 action 触发执行队列
                action_add_queue_tail(act);            }
        }
    }
}
```

另外，在 RC boot 阶段之后，Init 进程会显式的调用：

```
queue_builtin_action(queue_property_triggers_action, "queue_propetry_
triggers");
```

比较该系统中的 property 值是否为 RC 文件中“=”赋予的值，如不相等将该 action 加入执行队列。

接下来 Init 完成以下工作后进入主循环。

（1）Init 进程调用 load_565rle_image(…)，将/initlogo.rle 显示在屏幕上。这里没什么好说的，就是 map 出 framebuffer，然后把图贴上去。

（2）把触发条件为 init、early-boot、boot 的 section 执行掉，这将导致大量 property 值被设置。

（3）把触发条件为 property 等于某个值的 section 执行掉。

16.2.6　Init 的调试

Init 进程的信息输出并没有使用 Android 系统下常用的 LOG 输出功能。Init 进程的信息输出自成一套，Init 信息的出口是/dev/kmsg。在 Init 进程里错误、警告、调试等信息输出的工具如下：

```
#define ERROR(x…)   log_write(3, "<3>init: " x)
#define NOTICE(x…)  log_write(5, "<5>init: " x)
#define INFO(x…)    log_write(6, "<6>init: " x)
```

但是在 Init 进程通过 INFO 和 NOTICE 输出调试信息的时候却什么也得不到。输出的信息为什么会丢失？进一步分析 void log_write(…)：

```
void log_write(int level, const char *fmt, …)
{
    …
    /*这里是在 init 进程里通过 INFO 和 NOTICE 都无法调试输出的原因，Android 系统中
    log_level 为默认值 4，所以这里就返回了*/
    if (level > log_level) return;
    //如果没有 log_fd 文件，依然返回
    if (log_fd < 0) return;
    …
    //把内容全都写入 log_fd 文件
    write(log_fd, buf, strlen(buf));
}
```

我们发现信息的输出通过写 log_fd 这个文件来实现。那么 log_fd 是什么文件，在哪里打开呢？Init 的 main 函数里会调用 void log_init(void)函数来初始化自己的 LOG 文件。

```
void log_init(void)
{   …
    if (mknod(name, S_IFCHR | 0600, (1 << 8) | 11) == 0) { /*创建主设备号为 1、
    次设备号为 11 的字符文件*/
        log_fd = open(name, O_WRONLY);//以只写方式打开 log_fd 文件
    … }
}
```

可见 log_fd 文件对应的是 Linux 系统中主设备号为 1、次设备号为 11 的字符文件是/dev/kmsg，其实现位于内核源码树下 drivers/char/mem.c 文件。这个文件是内存文件，即不

存在于 nand 或磁盘，只是内核运行时虚拟的一个设备文件，其驱动实现如下：

```
static const struct file_operations kmsg_fops = {
    .write =    kmsg_write,
};
static ssize_t kmsg_write(struct file * file, const char __user * buf,
                size_t count, loff_t *ppos)
{
    …

    tmp = kmalloc(count + 1, GFP_KERNEL);   //在内核空间分配一块内存
    …
    if (!copy_from_user(tmp, buf, count)) {//将用户空间的内存复制到内核
        tmp[count] = 0;
        ret = printk("%s", tmp);            //把取到内容打出来
    …   }…
}
```

第 17 章　Interpreter 与 JIT

解释器是影响虚拟机性能关键因素，最初的 Dalvik 只有 C 语言版本的解释器，到汇编实现的 ASM 解释器，再到进一步将 JIT 做进解释器。在每一代 Android 版本的 release 中，都将提升 Dalvik 解释器的效率作为重要工作。

17.1　解释器编译结构

对于不同的处理器和指令集，Android 有着与之对应的高度优化 Interpreter 和 JIT 实现。为了支持这些不同的架构处理器和指令集，Android 使用了灵活的编译结构。

（1）在 dalvik/vm/Android.mk 里包含 ReconfigureDvm.mk。

（2）在 ReconfigureDvm.mk 里包含 Dvm.mk。

（3）dalvik/vm/Dvm.mk 是悬着指令集的关键，这里根据环境变量 dvm_arch_variant 选择指令集的对应实现。

对于集成 NEON 的 arm 处理器，对应的两个最关键的实现文件是 InterpAsm-armv7-a-neon.S 和 InterpC-armv7-a-neon.cpp。

```
mterp/out/InterpC-$(dvm_arch_variant).cpp.arm \
mterp/out/InterpAsm-$(dvm_arch_variant).S
```

makefile 里的包含的源文件是 InterpC-armv7-a-neon.cpp.arm，实际上没有这个文件，在 build/core/binary.mk 里面对 cpp_arm_sources 有着特殊的编译处理。InterpC-armv7-a-neon.cpp.arm 对应的就是 InterpC-armv7-a-neon.cpp。

17.2　Dalvik 寄存器编译模型

Dalvik 是基于寄存器的虚拟机，其寄存器编译模型是以函数为中心的，包括函数内部寄存器、函数间调用时参数寄存器与结果寄存器的分配与布局。

17.2.1　Callee 寄存器分配

Dalvik 的 Callee 函数寄存器分配规则如下：

对于没有产生调用的函数：

（1）设函数定义的局部变量为 n 则寄存器 0——寄存器（n–1）被依次分配给局部变量。

（2）寄存器 n 被分配给 this。

（3）设函数参数为 m，则寄存器（n+1）——寄存器（n+m）被依次分配给局部变量。

（4）如果函数运算过程中使用了中间变量，则为中间变量分配寄存器，寄存器号插入在 this 寄存器的后面。

以如下函数为例：

```
public int senix_register(int par1,int par2,int par3) {
        int local_var1=1;
        int local_var2=2;
        local_var1=local_var1+local_var2+par1+par2+par2;
             return local_var1;
        }
```

使用./out/host/linux-x86/bin/dexdump –d .odex 将字节码输出：

```
  name           :'senix_register' //函数名
     type        : '(III)I'//三个 int 参数，返回值也为 int
     access      : 0x0001 (PUBLIC) //属性
     code        -
     registers   : 7 //共用了 7 个寄存器
     ins         : 4 //输入变量
     outs        : 0//没有调用其他函数 outs 为 0
     insns size : 8 16bit code units
064efc:                                                          |[064efc]
com.android.launcher2.LauncherApplication.senix_register:(III)I
064f0c: 1210  |0000: const/4 v0, #int 1 // #1   /*local_var1=1;*/
064f0e: 1221  |0001: const/4 v1, #int 2 // #2   /*local_var2=2;*/
064f10: d802 0403  |0002: add-int/lit8 v2, v4, #int 3 // #03   /*local_var1
+local_var2 被优化掉成操作数#int 3，分配一个寄存器 v2 放置中间结果，这里实际效果是 v2=
local_var1+local_var2+par1*/
064f14: b052   |0004: add-int/2addr v2, v5   /* v2= local_var1+local_var2+
par1+par2 */
064f16: 9000 0205 |0005: add-int v0, v2, v5 /* v2= local_var1+local_var2+
par1+par2+par1，并且把结果放到寄存器 v0 中 */
064f1a: 0f00    |0007: return v0  /*以 v0 返回结果*/
     catches       : (none)
     positions     :
       0x0000 line=78
       0x0001 line=79
       0x0002 line=80
       0x0007 line=81
     locals        :
       0x0001 - 0x0008 reg=0 local_var1 I //寄存器 0 分配给 local_var1
       0x0002 - 0x0008 reg=1 local_var2 I //寄存器 1 分配给 local_var2
       0x0000 - 0x0008 reg=3 this Lcom/android/launcher2/Launcher
```

```
Application; /*寄存器 3 存放类对象 this，该函数是在 class LauncherApplication 里
添加的*/
        0x0000 - 0x0008 reg=4 par1 I  //寄存器 4 分配给输入参数 par4
        0x0000 - 0x0008 reg=5 par2 I  //寄存器 5 分配给输入参数 par5
        0x0000 - 0x0008 reg=6 par3 I  //寄存器 6 分配给输入参数 par5
```

17.2.2　Caller 寄存器分配

Caller 函数的寄存器分配规则是，首先满足 Callee 函数的寄存器分配规则，但是在此规则之外，产生调用时，要在 Callee 函数帧的高地址放入调用参数。这些调用参数被 Callee 当成 ins。

以如下函数为例分析：

```
  public int senix_register(int par1,int par2,int par3) {
        int local_var1=1;
        int local_var2=2;
        local_var1=local_var1+local_var2+par1+par2+par2;
             return local_var1;
       }
public int senix_register_caller(int par1,int par2,int par3) {
       int local_var1=1;
       int local_var2=senix_register(4,5,6);
       local_var1=local_var1+local_var2+par1+par2+par2;
                return local_var1;
       }
```

senix_register_caller(int par1,int par2,int par3)dump 的结果如下：

```
  name              : 'senix_register_caller'
    type            : '(III)I'
    access          : 0x0001 (PUBLIC)
    code            -
    registers       : 9  //共用了 9 个寄存器
    ins             : 4//3 个输入参数+this，this 不占寄存器
    outs            : 4//3 个输出参数+this，this 不占寄存器
    insns size      : 15 16-bit code units|[064f28
064f28:]
com.android.launcher2.LauncherApplication.senix_register_caller:(III)I
064f38: 1210  |0000: const/4 v0, #int 1 // #1
064f3a: 1242  |0001: const/4 v2, #int 4 // #4  /*给参数 4 分配寄存器 2*/
064f3c: 1253  |0002: const/4 v3, #int 5 // #5  /*给参数 4 分配寄存器 3*/
064f3e: 1264  |0003: const/4 v4, #int 6 // #6  /*给参数 6 分配寄存器 2*/
064f40: f840 7400 2543  |0004: +invoke-virtual-quick {v5, v2, v3, v4}, [0074]
// vtable #0074/* 调用发生了，3 个参数加 this */
064f46: 0a01   |0007: move-result v1  /* 把返回值放到 v1*/
```

```
/*下面为计算操作，参见上一节分析 */
064f48: 9002 0001                        |0008: add-int v2, v0, v1
064f4c: b062                             |000a: add-int/2addr v2, v6
064f4e: b072                             |000b: add-int/2addr v2, v7
064f50: 9000 0207                        |000c: add-int v0, v2, v7
064f54: 0f00                             |000e: return v0
      catches          : (none)
      positions        :
        0x0000 line=88
        0x0001 line=89
        0x0008 line=90
        0x000e line=91
      locals           :
        0x0001 - 0x000f reg=0 local_var1 I //寄存器 0 分配给 local_var1
        0x0008 - 0x000f reg=1 local_var2 I //寄存器 1 分配给 local_var2
        0x0000  -  0x000f   reg=5  this  Lcom/android/launcher2/Launcher
Application; /*寄存器 5 分配给 this*/
        0x0000 - 0x000f reg=6 par1 I//寄存器 6 分配给 par1
        0x0000 - 0x000f reg=7 par2 I//寄存器 7 分配给 par2
        0x0000 - 0x000f reg=8 par3 I//寄存器 8 分配给 par3
```

寄存器 v2、v3、v4 分配给三个调用参数。到了这里还是不能看清这些 outs 到底放在哪里。要解决这个问题需分析解释器处理 invokeMethod_XXX 指令时是如何安排参数的。

17.2.3 outs 的处理

分析 outs 的处理离不开对函数调用的分析，在解释器遇到 Dalvik 函数调用指令 invokeMethod 时，其 C 解释器 handler 如下（上文在分析 native 函数调用时遇到过这个 handler，这里侧重分析的寄存器布局）：

```
//C 解释器的 invokeMethodhandler
GOTO_TARGET(invokeMethod, bool methodCallRange, const Method* _method
ToCall,
    u2 count, u2 regs)
    {
        u4* outs;
        int i;
       /*首先正如注释所说，复制参数，这里分为两种情况，如果一个函数参数个数比较多，则变
        量 methodCallRange 成帧。对于每个参数，都用 GET_REGISTER 从当前函数帧里取值，
        再复制给参数在 Callee 函数中的寄存器*/

        if (methodCallRange) {
           //在当前栈上给参数分配空间，vsrc1 就是参数个数
            outs = OUTS_FROM_FP(fp, vsrc1);
            for (i = 0; i < vsrc1; i++)
```

```
                outs[i] = GET_REGISTER(vdst+i);
        } else {
            u4 count = vsrc1 >> 4;

          //在当前栈上给参数分配空间，count 就是参数个数
          outs = OUTS_FROM_FP(fp, count);
          assert((vdst >> 16) == 0);  // 16bits -or- high 16bits clear
          switch (count) {
         //根据参数个数的不同，有着不同的处理
          case 5:
              outs[4] = GET_REGISTER(vsrc1 & 0x0f);
         …
          case 1:
              outs[0] = GET_REGISTER(vdst & 0x0f);
          default:
              ;
          }
        }
    }
```

在完成了参数分配及初始化之后，分析参数空间的分配 OUTS_FROM_FP：

```
#define OUTS_FROM_FP(_fp, _argCount) \
    ((u4*) ((u1*)SAVEAREA_FROM_FP(_fp) - sizeof(u4) * (_argCount)))
#define SAVEAREA_FROM_FP(_fp)   ((StackSaveArea*)(_fp) -1)
```

不难发现，不是从当前帧指针_fp 往下分配，而是越过 StackSaveArea 在往下分出空间。这样做的原因是_fp 指针是用来索引寄存器，而一个函数帧里在_fp 往下还存在着一个 VM-specific internal goop-- struct StackSaveArea。由此可以得出以下结论。

（1）outs 其实就是从 Caller 角度看的调用参数，就是 Callee 的 ins。

（2）outs 到 ins 是拷贝，outs 分配的那些寄存器在 Caller 还可作为他用。

17.3　Portable Interpreter 结构

最初的几个 Android 版本里，Dalvik 的解释器是用 C 写的。这种解释器执行速度较慢，但可读性较强，移植性好，在以后 Android 版本里尽管实现了汇编优化的解释器，但这种 portable 解释器依然存在。在 Android 向某个全新架构的处理器上移植时是没有对应的汇编解释器的，这时 portable 的价值就体现出来了。

该解释器的核心是一个 handler 数组，其定义如下：

```
#define DEFINE_GOTO_TABLE(_name) \
    static const void* _name[kNumDalvikInstructions] = {                \
        …
```

```
        H(OP_MOVE_WIDE_FROM16),                              \
       …
        H(OP_MOVE_OBJECT_16),                                \
        H(OP_MOVE_RESULT),                                   \
        H(OP_MOVE_RESULT_WIDE),                              \
        H(OP_MOVE_RESULT_OBJECT),                            \
        H(OP_MOVE_EXCEPTION),
        …
        \ }
```

该数组 handlerTable 存放着每个操作码的 handler 地址，每遇到一个操作码就跳到这个数组里取出 handler 来执行该操作码。而每条操作码的 handler 结构定义如下：

```
HANDLE_OPCODE(OP_XXX)
    FINISH(…);
OP_END
```

其中 HANDLE_OPCODE(OP_XXX)被定义成该段 handler 标号，handlerTable 对应项指向这个地址，而每个操作码执行完，都通过 FINISH(…)取出下一个操作码，然后跳入该操作码对应的 handler 中。

```
# define FINISH(_offset) {                                  \
         //将 PC 指向下一条字节码
         ADJUST_PC(_offset);                                 \
         //取出字节码到 inst
        inst = FETCH(0);                                     \
        …
        //INST_INST(inst)即为指令编号，根据这个编号索引 handlerTable 的位置
        goto *handlerTable[INST_INST(inst)];                  \
    }
```

其中，ADJUST_PC(_offset)是将操作码代码段的 PC 指针指向下一个操作码，FETCH(0)的作用是取出这个操作码，最后跳入下一个操作码的 handler。

17.4 ASM Interpreter

Dalvik 默认的解释器就是这种汇编优化的解释器，根据不同 CPU 架构有不同的实现，本文主要讨论 ARM V7 架构的实现。

17.4.1 基本结构

与 portbale 解释器一样，ASM 解释器也是一个由不同字节码解释器组成的大数组，这是 ASM Interpreter 的 mainhandler，在正常运行时使用，其实现在文件 dalvik/vm/mterp/

out/InterpAsm-armv7-a-neon.S 中。

（1）mainHandler 大数组定义如下：

```
//大数组的基地址：dvmAsmInstructionStart。即为编号为 0 的 NOP 指令的地址
dvmAsmInstructionStart = .L_OP_NOP
//代码段
    .text
/*偏移量 64，每个 handler 64byte，不够的话再跳的其他地方，但要保证每个 handler 64
字节的入口*/
    .balign 64
.L_OP_NOP: /* 0x00 */
/* File: armv5te/OP_NOP.S */
  …
//64 字节对齐，第二个字节码 MOVE 的 handler
    .balign 64
.L_OP_MOVE: /* 0x01 */
/* File: armv6t2/OP_MOVE.S */
     /* for move, move-object, long-to-int */
     /* op vA, vB */
     mov     r1, rINST, lsr #12          @ r1<- B from 15:12
     ubfx    r0, rINST, #8, #4           @ r0<- A from 11:8
     FETCH_ADVANCE_INST(1)               @ advance rPC, load rINST
     …
    .balign 64
    .size   dvmAsmInstructionStart, .-dvmAsmInstructionStart
    .global dvmAsmInstructionEnd
    //mainhandler 数组结束
    dvmAsmInstructionEnd:
```

对于有些字节码不能用 64 字节完成其 handler 实现，ASM 解释器将其余实现放在代码段 dvmAsmSisterStart 里。

（2）ALThandler

ASM Interpreter 的还有一个 ALThandler，在 JIT 和 debugger 时使用，其实现也在文件 dalvik/vm/mterp/out/InterpAsm-armv7-a-neon.S 中。

```
//全局变量 dvmAsmAltInstructionStart
.global dvmAsmAltInstructionStart
    .type   dvmAsmAltInstructionStart, %function
//代码段
    .text
//ALThandler 数组也是跟字节码指令一一对应
dvmAsmAltInstructionStart = .L_ALT_OP_NOP
//也是 64 字节对齐
    .balign 64
.L_ALT_OP_NOP: /* 0x00 */
```

```
…
    .balign 64
.L_ALT_OP_MOVE: /* 0x01 */
…
    .balign 64
//详细分析一个 ALT handler 的结构，其余 ALThandler 类似
.L_ALT_OP_IF_GEZ: /* 0x3b */
//把线程结构的 breakFlags 放到 r3
    ldrb   r3, [rSELF, #offThread_breakFlags]
    adrl   lr, dvmAsmInstructionStart + (59 * 64)
    ldr    rIBASE, [rSELF, #offThread_curHandlerTable]
/*检查 breakFlags 是否为 0，如为 0 直接跳到 mainhandler，lr 为 mainhandler 数组里对
应地址*/
    cmp    r3, #0
    bxeq   lr                  @ nothing to do - jump to real handler
    EXPORT_PC()
    mov    r0, rPC              @ arg0
    mov    r1, rFP              @ arg1
mov    r2, rSELF            @ arg2
//breakFlags 被置位，需要进一步到 dvmCheckBefore 检查
    b      dvmCheckBefore       @ (dPC,dFP,self) tail call
     …
.balign 64
//althander 数组长度
.size   dvmAsmAltInstructionStart, .-dvmAsmAltInstructionStart
.global dvmAsmAltInstructionEnd
     //althander 数组结束
dvmAsmAltInstructionEnd:
```

（3）Handler 的启用

在一个 Dalvik 创建之初，在线程的管理结构里记录下该 handler 的地址。

```
static Thread* allocThread(int interpStackSize)
{   …
    //mainHandlerTable 偏移值为 88，即为：offThread_mainHandlerTable
    thread->mainHandlerTable = dvmAsmInstructionStart;
    // altHandlerTable 即为 dvmAsmAltInstructionStart;
    thread->altHandlerTable = dvmAsmAltInstructionStart;
    //interpBreak.ctl.curHandlerTable 偏移值为 40，即为 offThread_
    curHandlerTable
    thread->interpBreak.ctl.curHandlerTable = thread->mainHandlerTable;
    …
}
```

17.4.2 运行时模型与基本操作

（1）ASM 解释器定义了专门的寄存器来对应 Dalvik 虚拟机模型。

```
//rPC 指向 dalvik 操作码的地址
#define rPC     r4
//fFP 指向 dalvik 的帧，这是在编译时确定下来的寄存器组
#define rFP     r5
//rSELF 指向当前线程的 struct Thread 结构
#define rSELF   r6
//rINST 为当前指令
#define rINST   r7
//rIBASE 指向字节码 handler 大数组的基地址
#define rIBASE  r8
```

（2）基本操作如下：

```
//把当前操作码作为基址，偏移量为_countX2，开始的无符号半字加载到寄存器_reg 中
#define FETCH(_reg, _count)     ldrh    _reg, [rPC, #((_count)*2)]
//把 rPC 和 rFP 从当前线程的 struct Thread 结构里取出来
#define LOAD_PC_FP_FROM_SELF()  ldmia   rSELF, {rPC, rFP}
//把以_vreg 为索引的寄存器加载在_reg 中，_vreg 的索引值以 rFP 为基准
#define GET_VREG(_reg, _vreg)   ldr     _reg, [rFP, _vreg, lsl #2]
//从 struct InterpSaveState 取出字节码指令地址与帧地址放入 rPC 和 rFP
#define LOAD_PC_FP_FROM_SELF()  ldmia   rSELF, {rPC, rFP}

//跳到_reg 字节码对应的 handler，因为与 64 字节对齐，所以 lsl#6
#define GOTO_OPCODE(_reg)       add     pc, rIBASE, _reg, lsl #6
```

17.4.3　ASM Interpreter 入口

dvmMterpStdRun 是解释器入口，不同的解释器有着不同的实现，对于 ASM Interpreter，不仅要满足 C to ASM 调用规范，而且承接好 DVM 虚拟机 Context，　其实现如下：

```
//ASM 版解释器入口
dvmMterpStdRun:
#define MTERP_ENTRY1 \
    .save {r4-r10,fp,lr}; \
    stmfd   sp!, {r4-r10,fp,lr}
#define MTERP_ENTRY2 \
    .pad    #4; \
    sub     sp, sp, #4

    /*引用上文定义宏，保存 r4-r10,fp,lr 寄存器*/
    .fnstart
    MTERP_ENTRY1
    MTERP_ENTRY2

    /* 保存栈指针 */
```

```
    str     sp, [r0, #offThread_bailPtr]

    /* r0 里存放当前线程的 struct Thread */
    mov     rSELF, r0
/* 从当前线程 struct Thread 的 struct InterpSaveState 里取出字节码地址放入 fPC，
   帧地址放入 rFP */
    LOAD_PC_FP_FROM_SELF()
    /* rIBASE 就是 handler 数组的地址*/
    ldr     rIBASE, [rSELF, #offThread_curHandlerTable] @ set rIBASE

#if defined(WITH_JIT)
/*jit 功能 enable 时的处理*/
.LentryInstr:
  /* Entry is always a possible trace start */
  /*把 struct Thread 的 pJitProfTable;放入 r0，JitProfTable 是用来统计热点的阈值
  表*/
  ldr     r0, [rSELF, #offThread_pJitProfTable]
  //取出当前指令
     FETCH_INST()
     mov     r1, #0                      @ prepare the value for the new state
  str     r1, [rSELF, #offThread_inJitCodeCache] @ back to the interp land
  /*如果 pJitProfTable 为 0 就表示没有热点检测，自然就没有 jit 这回事了*/
     cmp     r0,#0                       @ is profiling disabled?
#if !defined(WITH_SELF_VERIFICATION)
/*入口也是种跳转，去检测是否热点，是否需要 jit*/
    bne     common_updateProfile @ profiling is enabled
#else
    …
#endif
/*这是 jit eable 时有效的编译，第一条字节码已经被取出到 rINST，把 rINST 里的指令编码
  取出来放到寄存器 ip 里 */
1:
GET_INST_OPCODE(ip)
/*根据寄存器 ip 的值，跳转到第一条字节码对应的 handler，至此 ASM Interpreter 启动了，
  以后每条字节码都会取出其后的字节码，并跳入对应的 handler */
    GOTO_OPCODE(ip)
#else
    /* start executing the instruction at rPC */
    FETCH_INST()                        @ load rINST from rPC
    GET_INST_OPCODE(ip)                 @ extract opcode from rINST
    GOTO_OPCODE(ip)                     @ jump to next instruction
#endif
```

17.5 Interpreter 的切换

本节分析 Dalvik 虚拟机是通过何种方式选择解释器的。

(1) 查找系统属性里解释器执行模式。

```
int AndroidRuntime::startVm(JavaVM** pJavaVM, JNIEnv** pEnv)
{
   …
//取出系统属性 dalvik.vm.execution-mode
    property_get("dalvik.vm.execution-mode", propBuf, "");
    if (strcmp(propBuf, "int:portable") == 0) {
    /*portable 解释器，这个是标配，但是不是默认使用的解释器。在有些新的架构处理器上没
    有别的解释器可用，可以先用这个*/
        executionMode = kEMIntPortable;
} else if (strcmp(propBuf, "int:fast") == 0) {
/* executionMode=2，所谓 fast 解释器就是汇编优化过但是没有 JIT 功能的解释器，JIT 对
   于小系统有可能带来副作用，这个解释器是不错选择*/
        executionMode = kEMIntFast;
#if defined(WITH_JIT)
     } else if (strcmp(propBuf, "int:jit") == 0) {
     /*executionMode=3，既汇编优化又带 JIT 功能。如果是 ARM v7 以上的机器，就是这个了*/
        executionMode = kEMJitCompiler;
#endif
    }
    …
}
```

(2) 解释器选择。

根据执行模式选择解释器。在解释器入口处有很多处理，这里仅关注解释器选择。

```
void dvmInterpret(Thread* self, const Method* method, JValue* pResult)
{   …
    if (gDvm.executionMode == kExecutionModeInterpFast)
    // dvmMterpStd 是 ASM 优化但不带 JIT 功能的实现
        stdInterp = dvmMterpStd;
#if defined(WITH_JIT)
    else if (gDvm.executionMode == kExecutionModeJit)
    /*dvmMterpStd 是 ASM 优化但带 JIT 功能的实现，与上者的区别在编译选项 WITH_JIT 是
      否激活 */
        stdInterp = dvmMterpStd;
#endif
   Else
   //Portable 解释器，入口函数是 void dvmInterpretPortable(…)
```

```
        stdInterp = dvmInterpretPortable;
    ...
}
```

17.6 Dalvik 运行时帧结构

在 Dalvik 运行时，每个函数有自己的 Frame，首先分析 Dalvik 源码里对 Frame 结构的描述：

```
Low addresses (0x00000000)

+- - - - - - - - -+
-  out0           -
+-----------------+  <-- stack ptr (top of stack)
+  VM-specific    +
+  internal goop  +
+-----------------+  <-- curFrame: FP for cur function
+  v0 == local0   +
+-----------------+  +-----------------+
+  out0           +  +  v1 == in0      +
+-----------------+  +-----------------+
+  out1           +  +  v2 == in1      +
+-----------------+  +-----------------+
+  VM-specific    +
+  internal goop  +
+-----------------+  <-- frame ptr (FP) for previous function
+  v0 == local0   +
+-----------------+
+  v1 == local1   +
+-----------------+
+  v2 == in0      +
+-----------------+
+  v3 == in1      +
+-----------------+
+  v4 == in2      +
+-----------------+
-                 -
-                 -
-                 -
+-----------------+  <-- interpStackStart

    High addresses (0xffffffff)
```

其中寄存器分配规则是由 Dalvik 编译器决定的，而 VM-specific internal goop 就是 struct

StackSaveArea，该结构定义如下：

```
struct StackSaveArea {
   …
   u4*        prevFrame;
   //该 PC 指针位并不是二进制的 r15，而是当前字节码地址
   const u2*   savedPc;
   //对应的函数结构指针
   const Method* method;
    union {
        u4          localRefCookie;
        /*保存当前字节码的地址到其成员变量，通常在 EXPORT_PC()时发生*/
        const u2*   currentPc;
    } xtra;
    …
};
```

字节码的地址保存操作如下：

```
#define EXPORT_PC() \
    str rPC, [rFP, #(-sizeofStackSaveArea + offStackSaveArea_ currentPc)]
```

其中 offStackSaveArea_currentPc 被定义为 12，sizeofStackSaveArea 即为 struct StackSaveArea 结构的 size。rFP 为当前帧基地址，参考 17.5 节的 Frame 结构，[rFP, #(-sizeofStackSaveArea + offStackSaveArea_currentPc)]为 xtra. currentPc 所在地址。

17.7　JIT

JIT 的方式可以函数为单位也可以 trace 为单位。前者整体编译一个函数，后者编译一个热点路径。Dalvik 目前工作方式采用第二种，而实际上 Android 中已经有了第一种方式的代码，但是在 Android 2.3 版本中并未激活。本章以 ASM Interpreter 为基础分析 JIT 机制。

17.7.1　热点检测

JIT 的第一步是监测热点，即找到最频繁执行的 trace 或函数，而进入热点检测之前，ASM Interpreter 准备好如下数据：

```
//当前线程 struct Thread 的 pJitProfTable 放入 r0
r0    <= pJitProfTable (verified non-NULL)
//当前字节码地址放入 rPC
rPC    <= Dalvik PC
// rINST 为下一条字节码指令
rINST <= next instruction
```

其中 JitProfTable 在 bool compilerThreadStartup(void)里被创建，里面每一项对应 hash 到此字节码的 hot 程度，初始设置为 gDvmJit.threshold（gDvmJit.threshold 针对不同架构有着不同的默认值，对于 v5te 默认值为 200，对于 v7 架构默认值为 40），每当代码执行到此一次减一，当减为 0 时做 JIT。

热点检测跟 asm interpreter 一样，实现在文件都在 dalvik/vm/mterp/out/InterpAsm-armv7-a-neon.S 中。

```
common_updateProfile:
//根据 rPC 值算出 hash 值，再根据 hash 值找出 JitProfTable 对应项
eor     r3,rPC,rPC,lsr #12 @ cheap, but fast hash function
lsl     r3,r3,#(32 - JIT_PROF_SIZE_LOG_2)         @ shift out excess bits
ldrb    r1,[r0,r3,lsr #(32 - JIT_PROF_SIZE_LOG_2)] @ get counter
//把 rINST 字节码的编号放到寄存器 ip
GET_INST_OPCODE(ip)
//将 JitProfTable 对应项减 1
subs    r1,r1,#1          @ decrement counter
//将结果存回 JitProfTable 对应项
strb    r1,[r0,r3,lsr #(32 - JIT_PROF_SIZE_LOG_2)] @ and store it
//如果没到做 jit 阈值，返回 asm interpreter
GOTO_OPCODE_IFNE(ip)       @ if not threshold, fallthrough otherwise
//到了这里说明 rPC 指向的解码执行足够频繁，需要做 jit 了
/* Looks good, reset the counter */
//取出阈值，并存入 JitProfTable 对应项
ldr     r1, [rSELF, #offThread_jitThreshold]
strb    r1,[r0,r3,lsr #(32 - JIT_PROF_SIZE_LOG_2)] @ reset counter
// SAVEAREA_FROM_FP(fp)->xtra.currentPc = pc
EXPORT_PC()
mov     r0,rPC
mov     r1,rSELF
//去 jit 过的代码段里查找当前 rPC 地址的字节码段是否存在
bl      dvmJitGetTraceAddrThread    @ (pc, self)
//如果当前 rPC 地址处被 jit 过，r0 里是返回地址，存入 inJitCodeCache
str     r0, [rSELF, #offThread_inJitCodeCache] @ set the inJitCodeCache flag
mov     r1, rPC                     @ arg1 of translation may need this
mov     lr, #0                      @  in case target is HANDLER_INTERPRET
//检查是否有 jit 过的二进制代码
    cmp     r0,#0
#if !defined(WITH_SELF_VERIFICATION)
    //直接跳到 jit 过的二进制代码
    bxne    r0                      @ jump to the translation
    //rPC 处代码还没做过 jit
    mov     r2,#kJitTSelectRequest      @ ask for trace selection
    @ fall-through to common_selectTrace
#else
```

```
 //实际设备上这个编译选项是 disable，不考虑这种 WITH_SELF_VERIFICATION 情况
#endif
common_selectTrace:
    /*struct Thread 的 interpBreak.ctl.subMode 记录着当前 jit 的模式*/
    ldrh    r0,[rSELF,#offThread_subMode]
    /*测试标志位 kSubModeJitTraceBuild 和 kSubModeJitSV 若不为零，说明当前已经激活
      jit。值得说明的是 jit 工作的本身是在另外一个线程里进行。激活 jit 意味着当前 dalvik
      线程向 jit 工作线程提交了 jit 工作申请*/
    ands    r0, #(kSubModeJitTraceBuild | kSubModeJitSV)
    //如果已经激活 jit，继续执行当前 Dalvik 线程即可
    bne     3f                      @ already doing JIT work, continue
    //r2 里已经放入 kJitTSelectRequest
        str     r2,[rSELF,#offThread_jitState]
    mov     r0, rSELF
        EXPORT_PC()
    SAVE_PC_FP_TO_SELF()                 @ copy of pc/fp to Thread
    /*进入 jit 激活操作，r0 里是当前线程的 struct Thread，作为 void
      dvmJitCheckTraceRequest(Thread* self)的参数，该函数将导致
      curHandlerTable 切换成 ALThandlerTable*/
        bl      dvmJitCheckTraceRequest
    3:
        FETCH_INST()
        //重新取出 handlertable 基地址，这时已经发生了改变，切换成 ALThandlerTable
           ldr    rIBASE, [rSELF, #offThread_curHandlerTable]
    4:
    //继续 Dalvik 当前线程
    GET_INST_OPCODE(ip)                 @ extract opcode from rINST
    GOTO_OPCODE(ip)
    /* no return */
#endif
```

17.7.2　Mode 切换

在进入该函数之前，struct Thread 的 jitState 被置位为 kJitTSelectRequest，代码如下：

```
void dvmJitCheckTraceRequest(Thread* self)
{   …
    switch (self->jitState) {
            /*jitState 为 kJitTSelectRequest 时该分支有效，该状态在 common_
              selectTrace:处被设置*/
          case kJitTSelectRequest:
          /*jitstate 状态转换成 kJitTSelect，dvmcheckjit 里将处理这个状态*/
          self->jitState = kJitTSelect;
   /* kSubModeJitTraceBuild 为 0x4000，这里将导致 handlertable 的切换*/
         dvmEnableSubMode(self, kSubModeJitTraceBuild);
```

```
        …
      }
}
```

调用层次如下：

```
void dvmEnableSubMode(Thread* thread, ExecutionSubModes subMode)
{   //参数 subMode 为 kSubModeJitTraceBuild =0x4000
    updateInterpBreak(thread, subMode, true);
}
//参数 subMode 为 kSubModeJitTraceBuild =0x4000，enable 有效
void updateInterpBreak(Thread* thread, ExecutionSubModes subMode, bool
enable)
{   …
    do {
        …
        if (enable)
            newValue.ctl.subMode |= subMode;
        else
            newValue.ctl.subMode &= ~subMode;
/* newValue.ctl.subMode 已经被置位为 kSubModeJitTraceBuild，而 SINGLESTEP_
   BREAK_MASK 有效位包括 kSubModeJitTraceBuild/
        if (newValue.ctl.subMode & SINGLESTEP_BREAK_MASK)
            newValue.ctl.breakFlags |= kInterpSingleStep;
        …
/*因为 newValue.ctl.breakFlags 被置位为 kInterpSingleStep，这里导致 asm
interpreter handlertable 切换到 althandlertable*/
        newValue.ctl.curHandlerTable = (newValue.ctl.breakFlags) ?
            thread->altHandlerTable : thread->mainHandlerTable;
      } while (
/*64 位操作，将修改后的值更新给 union InterpBreak*/
       dvmQuasiAtomicCas64(oldValue.all, newValue.all,
       &thread-> interpBreak. all) != 0); }
}
```

17.7.3 JIT 提交

JIT 时需要使用 ALThandler，其结构前面已做过分析，这里有两点需要明确。

（1）具体工作还是由 mainhandlertable 处理。

（2）每次将处理递交给 mainhandlertable 之前都要调用 void dvmCheckBefore(…)，检测当前状态并提交 JIT 申请。

```
void dvmCheckBefore(const u2 *pc, u4 *fp, Thread* self)
{…
#if defined(WITH_JIT)
```

```
    // self->interpBreak.ctl.subMode 被置位 kSubModeJitTraceBuild
    if (self->interpBreak.ctl.subMode &
            (kSubModeJitTraceBuild | kSubModeJitSV)) {
        if (self->interpBreak.ctl.subMode & kSubModeJitTraceBuild) {
          /*vid dvmCheckJit(…)实现具体的 jit 递交工作*/
            dvmCheckJit(pc, self);
        }
    }
#endif
…
}
```

17.8　Compile

Compile 是指 JIT 的编译阶段，这里完成 Dalvik 指令到机器指令的编译动作。这个工作分为两部分，首先解释器完成 tracerun 的分析之后，然后 Dalvik 编译线程完成编译工作。

17.8.1　基础数据结构

Compile 其实就是一个编译器的后端，出现很多不同于虚拟机的基础设施，所以在进入到 Compile 之前，首先分析 JIT Compile 使用到的基本数据结构及其概念。

（1）struct DecodedInstruction 用来表示一条 Dalvik 指令。

Dalvik 操作码被分成 36 种模式，如 op、op vA, vB、op vA, #+B、op vAA, vBB, vCC 等。该结构分解出该操作码的操作数和操作码。

```
struct DecodedInstruction {
    u4      vA;         //vA、vB、vC 都是分解出来的操作数
    u4      vB;
    u8      vB_wide;        /* for kFmt51l */
    u4      vC;
    u4      arg[5];         /* vC/D/E/F/G in invoke or filled-new-array */
    Opcode  opcode;     //操作码
    InstructionIndexType indexType;
};
```

（2）struct JitTraceRun 用来记录一个 tracerun 对应的字节码起始地址、该 tracerun 包含的字节码长度等信息，或者用来记录元数据信息。

```
struct JitTraceRun {
union {
   JitCodeDesc frag; //trace run 描述
   void*      meta;//若用来描述元数据，该指针指向元数据地址
} info;
```

```
    u4 isCode:1;//通过这一位来辨别是否是元数据
    u4 unused:31;
};
```

（3）typedef struct MIR 表示一条 Dalvik 指令编译结构。

```
typedef struct MIR {
    //上述字节码描述结构
    DecodedInstruction dalvikInsn;
    //Dalvik 指令长度
    unsigned int width;
    unsigned int offset;
    /*一个 tracerun 的 Dalvik 指令通过如下两个成员变量 prev 和 next 挂在对应的 struct
     BasicBlock 结构上*/
    struct MIR *prev;
    struct MIR *next;
    //SSA
    struct SSARepresentation *ssaRep;
    …
} MIR;
```

（4）struct BasicBlock 表示一个 TraceRun，除了从字节码段分析出的 TraceRun、entry 等，一些编译器加入的代码也用这个结构来表示。

```
typedef struct BasicBlock {
     …
     //本基本块起始地址
    unsigned int startOffset;
    …
     /*用来串自己的 typedef struct MIR 链表*/
    MIR *firstMIRInsn;
     MIR *lastMIRInsn;
     //紧邻的基本块
     struct BasicBlock *fallThrough;
     //跳转或调用的基本块
     struct BasicBlock *taken;
    …
    //执行到本基本块前面的基本块
    BitVector *predecessors;
    …
    struct {
        BlockListType blockListType;
        GrowableList blocks;       //盛放本基本块的容器
    } successorBlockList;
} BasicBlock;
```

17.8.2　dalvik 指令格式分析

本节分析 Dalvik 代码格式分解的基本例程。

（1）取操作码。

```
Opcode dexOpcodeFromCodeUnit(u2 codeUnit) {
   //取出低 8 位
   int lowByte = codeUnit & 0xff;
   if (lowByte != 0xff) {
   /*如果低 8 位不全为 1，低 8 位就是操作码*/
       return (Opcode) lowByte;
   } else {
   /*否则高 8 位有效，opcode 为高 8 位+0x100*/
       return (Opcode) ((codeUnit >> 8) | 0x100);
    }
}
```

（2）指令宽度、类型属性。

等到了指令操作码以后，指令宽度、格式、属性等信息直接查表即可。Dalvik 准备了 4 张表。

```
InstructionInfoTables gDexOpcodeInfo = {
    gInstructionFormatTable,
    gInstructionIndexTypeTable,
    gOpcodeFlagsTable,
    gInstructionWidthTable
};
```

如果要知道指令宽度，查表 gInstructionWidthTable。

```
size_t dexGetWidthFromOpcode(Opcode opcode)
{   …
    return gDexOpcodeInfo.widths[opcode];
}
```

要知道指令格式，查表 gInstructionFormatTable。

```
InstructionFormat dexGetFormatFromOpcode(Opcode opcode)
{   …
    return (InstructionFormat) gDexOpcodeInfo.formats[opcode];
}
```

17.8.3　TraceRun 分析

进入到 TraceRun 的背景时，ASM 解释器侦测到 JIT 的发生，切换到 ALThandlerTable，

而一旦切换到 ALThandlerTable 以后 TraceRun 的分析就开始了。

```
void dvmCheckJit(const u2* pc, Thread* self)
{    …
    // lastPC 是上一条字节码的地址，第一条字节码进来的时候 lastPC 为 NULL
    const u2 *lastPC = self->lastPC;
    //当前字节码地址
    self->lastPC = pc;

    switch (self->jitState) {
        int offset;
        DecodedInstruction decInsn;
        case kJitTSelect:
        //在开始做 Tracerun 时 jitState 为 kJitTSelect
           if (lastPC == NULL) break;
           /*Grow the trace around the last PC if jitState is kJitTSelect */
           dexDecodeInstruction(lastPC, &decInsn);

            /*如果遇到 OP_PACKED_SWITCH 和 OP_SPARSE_SWITCH 指令说明 TraceRun 分析
              结束 */
            if (self->totalTraceLen != 0 &&
                (decInsn.opcode == OP_PACKED_SWITCH ||
                 decInsn.opcode == OP_SPARSE_SWITCH)) {
                self->jitState = kJitTSelectEnd;
                break;
            }
            //flags 是前一条指令的属性
            flags = dexGetFlagsFromOpcode(decInsn.opcode);
            //len 是前一条指令的宽度
            len = dexGetWidthFromInstruction(lastPC);
            // offset 是前一条指令的偏移函数起始位置的长度
            offset = lastPC - self->traceMethod->insns;
            …
          /*根据指令流的连续性来判定是否为一个新 TraceRan 的开始*/
            if (lastPC != self->currRunHead + self->currRunLen) {
                /*出现不连续的指令流，新 TraceRun 开始*/
                int currTraceRun;
               /* self->currTraceRun 标记当前 tracerun 号*/
                /* We need to start a new trace run */
                currTraceRun = ++self->currTraceRun;
                /*新的 tracerun 开始了，currRunLen 置位为 0，currRunHead 为起始字节
                  码地址*/
                self->currRunLen = 0;
                self->currRunHead = (u2*)lastPC;
                 /*记下该 tracerun 的相关信息*/
```

```
        self->trace[currTraceRun].info.frag.startOffset = offset;
        …
        self->trace[currTraceRun].isCode = true;
    }
  /*当前 tracerun，指令个数增加*/
   self->trace[self->currTraceRun].info.frag.numInsts++;
   /*本次跟踪所有 tracerun 的指令个数增加*/
   self->totalTraceLen++;
   /*当前 tracerun 的最新地址*/
   self->currRunLen += len;
   /*
    * If the last instruction is an invoke, we will try to sneak in
    * the move-result* (if existent) into a separate trace run
    */
   /*代码注释很清楚：如果 lastPC 指令是函数调用类指令，则 tracerun 的个数要减
   一，以便能够塞进一个 move-result*/
   {
     //指令 gOpcodeFlagsTable 表里指出是函数调用类指令
     int needReservedRun = (flags & kInstrInvoke) ? 1 : 0;
     /*一次 tracerun 的个数被定为 64，如果超过了 64 即使还有 tracerun 也不在
       这次做了*/
     if (self->currTraceRun ==
          (MAX_JIT_RUN_LEN - 1 - needReservedRun)) {
     //收工
       self->jitState = kJitTSelectEnd;
     }
   }
  /*如果不是无条件跳转指令，并且指令属性里有 kInstrCanBranch、
    kInstrCanSwitch、kInstrCanReturn、kInstrInvoke 属性，说明本次
    tacerun 的跟踪应该结束了*/
   if (!dexIsGoto(flags) &&
         ((flags & (kInstrCanBranch |
                    kInstrCanSwitch |
                    kInstrCanReturn |
                    kInstrInvoke)) != 0)) {
          self->jitState = kJitTSelectEnd;
        …
        //对于函数调用的处理
        if (flags & kInstrInvoke) {
        //用 3 个 struct JitTraceRun 放函数调用的元数据信息
          insertClassMethodInfo(self, thisClass, curMethod,
          &decInsn);
     //加一个 tracerun，用来结束返回
          insertMoveResult(lastPC, len, offset, self);
        }
```

```
            }
           //如果遇到了 THROW，收工
            if ((decInsn.opcode == OP_THROW) || (lastPC == pc)){
                self->jitState = kJitTSelectEnd;
            }
            //无论是什么情况，如果超过了 64 个 tracerun，收工
            if (self->totalTraceLen >= JIT_MAX_TRACE_LEN) {
                self->jitState = kJitTSelectEnd;
            }
           ...
        case kJitTSelectEnd:
        //tracerun 的分析收工了，下一步就是向 compiler 线程提交编译申请
            {
                ...
                //把前面收集到的 tracerun 信息打包在 JitTraceDescription 里
                JitTraceDescription* desc =
                   (JitTraceDescription*)malloc(sizeof(JitTraceDescription)+
                     sizeof(JitTraceRun) * (self->currTraceRun+1));
                 ...
                //向 compiler 线程提交编译申请
                if (dvmCompilerWorkEnqueue(
                       self->currTraceHead,kWorkOrderTrace,desc)) {
                    /*入队*/
                    if (gDvmJit.blockingMode) {
                        dvmCompilerDrainQueue();
                    }
                } else {
                    ...
                }
                /*尽管 compile 的工作还没确定完成，但是对于 ASM 解释器，jit
                的工作到此已经完成了*/
                self->jitState = kJitDone;
                allDone = true;
            }
            break;
        case kJitDone:
           // allDone 恢复为 true
            allDone = true;
            break;
            ...
    }
...
     if (allDone) {
        //这里导致 ASM 解释器切换到 mainhandlertable
         dvmDisableSubMode(self, kSubModeJitTraceBuild);
```

```
        }
        /*从这里返回 mianhandlertable，再也没有 tracerun 的检查，ASM 解释器恢复正常状态*/
        return;
    }
```

17.8.4　MIR

MIR 即为 middle-level intermediate representation，在完成对 trace 之后，要对收集到的 traceruns 做逻辑分析，分割出基本块，其中 Dalvik 指令以 MIR 形式链接起来。

```
bool dvmCompileTrace(JitTraceDescription *desc, int numMaxInsts,
                     JitTranslationInfo *info, jmp_buf *bailPtr,
                     int optHints)
{
/*参数 JitTraceDescription *desc 里面是收集到的 traceruns 信息*/
/* dexCode 是这些 traceruns 所在的文件 */
const DexCode *dexCode = dvmGetMethodCode(desc->method);
/*首先定位到第一个 tracerun*/
const JitTraceRun* currRun = &desc->trace[0];
/*第一个 tracerun 偏移地址*/
    unsigned int curOffset = currRun->info.frag.startOffset;
unsigned int startOffset = curOffset;
/*第一个 tracerun 含有的字节码数目*/
unsigned int numInsts = currRun->info.frag.numInsts;
/* codePtr 直接指向第一个 tracerun 在 map 出的 odex 文件地址*/
const u2 *codePtr = dexCode->insns + curOffset;
…
/*本次编译结构描述*/
CompilationUnit cUnit;
GrowableList *blockList;
/*把编译信息收集到 cUnit 里*/
cUnit.method = desc->method;
cUnit.traceDesc = desc;
cUnit.jitMode = kJitTrace;
…
/*每次 trace 都包含若干 struct BasicBlock，本次 trace 所有的 struct BasicBlock 都
放在 cUnit.blockList 里记录 */
    blockList = &cUnit.blockList;
    dvmInitGrowableList(blockList, 8);
     …
    /* entry block 这是编译器自行加入的特殊 struct BasicBlock*/
    curBB = dvmCompilerNewBB(kEntryBlock, numBlocks++);
    dvmInsertGrowableList(blockList, (intptr_t) curBB);
    curBB->startOffset = curOffset;
    /* DalvikByteCode 这是与 tracerun 对应的 struct BasicBlock，每个 tracerun 都
```

```
有自己的 struct BasicBlock，当前这个 entryCodeBB 对应第一个 tracerun */
entryCodeBB = dvmCompilerNewBB(kDalvikByteCode, numBlocks++);
dvmInsertGrowableList(blockList, (intptr_t) entryCodeBB);
entryCodeBB->startOffset = curOffset;
/* fallThrough 表示 struct BasicBlock 间的逻辑关系为顺序执行，entry block 后
   的第一个顺序执行 struct BasicBlock 为第一个 tracerun*/
curBB->fallThrough = entryCodeBB;
/* entry block 不需要生成 MIR，curBB 从第一个 tracerun 开始*/
curBB = entryCodeBB;
while (1) {
/*每个字节码都对应一个 MIR */
    MIR *insn;
    /*字节码的宽度 */
    int width;
    /*为当前字节码分配 MIR */
    insn = (MIR *)dvmCompilerNew(sizeof(MIR), true);
    /*offset 指向当前字节码地址 */
    insn->offset = curOffset;
    /*把当前字节码宽度分析出来 */
    width = parseInsn(codePtr, &insn->dalvikInsn, cUnit.printMe);

    insn->width = width;
    traceSize += width;
    /*把当前 MIR 挂到当前 struct BasicBlock 的链表中，根据字节码的先后有严格的顺
    序关系 */
    dvmCompilerAppendMIR(curBB, insn);
    cUnit.numInsts++;
     /*查表取出当前字节码的属性*/
    int flags = dexGetFlagsFromOpcode(insn->dalvikInsn.opcode);

  /*invoke 类指令的处理*/
  if (flags & kInstrInvoke) {
 /*对于 invoke 类指令，在当前描述编译信息的 struct JitTraceRun 结构后面，连续放
   着描述类信息、类加载、当前函数的元数据信息，也是用 struct JitTraceRun 结构来
   表示。这些信息在 trace 分析时就被收集好，参见 void dvmCheckJit(…)*/
        /*当前函数信息*/
        const Method *calleeMethod = (const Method *)
            currRun[JIT_TRACE_CUR_METHOD].info.meta;
        …
        CallsiteInfo *callsiteInfo =
            (CallsiteInfo *)dvmCompilerNew(sizeof(CallsiteInfo), true);
       /*当前类描述信息*/
        callsiteInfo->classDescriptor = (const char *)
            currRun[JIT_TRACE_CLASS_DESC].info.meta;
        /*当前类加载器*/
```

```
            callsiteInfo->classLoader = (Object *)
                currRun[JIT_TRACE_CLASS_LOADER].info.meta;
            callsiteInfo->method = calleeMethod;
            insn->meta.callsiteInfo = callsiteInfo;
        }
        …
        /*完成了一个字节码的处理，numInsts 递减*/
        if (--numInsts == 0) {
        /*当前 tracerun 的字节码处理完毕，遇到两种情况*/
            if (currRun->info.frag.runEnd) {
            /*所有的 tracerun 处理完毕*/
                break;
            } else {
               /*还有 tracerun 待处理 */
                do {
               /* currRun 指向下一个 tracerun*/
                    currRun++;
                } while (!currRun->isCode);
                …
           /*为新的 tracerun 分配一个 DalvikByteCode 类型的 struct BasicBlock，用
             来记录这个 tracerun 的 MIR*/
                curBB = dvmCompilerNewBB(kDalvikByteCode, numBlocks++);
                dvmInsertGrowableList(blockList, (intptr_t) curBB);
               /*为新的 tracerun 更新起始地址、字节码数目、odex 文件地址等信息*/
                curOffset = currRun->info.frag.startOffset;
                numInsts = currRun->info.frag.numInsts;
                curBB->startOffset = curOffset;
                codePtr = dexCode->insns + curOffset;
            }
        } else {
         /*这是一个 tracerun 内的处理，字节码指针更新*/
            curOffset += width;
            codePtr += width;
        }
    }
…
}
```

17.8.5　基本块的逻辑关系

在完成基本块 MIR 链表生成之后，要分析出基本块之间的关系，这部分工作仍在 bool dvmCompileTrace(…)中进行，代码如下：

```
bool dvmCompileTrace(JitTraceDescription *desc, int numMaxInsts,
                     JitTranslationInfo *info, jmp_buf *bailPtr,
```

```
                         int optHints)
{   …
  /*这里依次取出基本块,然后根据每个基本块最后一个字节码指出的地址和其他基本块的起始地
    址对比来判决基本块之间的关系*/
   size_t blockId;
   for (blockId = 0; blockId < blockList->numUsed; blockId++) {
     /*根据基本块的索引依次取出基本块*/
       curBB = (BasicBlock *) dvmGrowableListGetElement(blockList,
       blockId);
     /*取出基本块最后一个字节码，前面介绍过，基本块的 MIR 链表是有着严格的顺序关系，
       lastMIRInsn 即为最后一个字节码的 MIR*/
       MIR *lastInsn = curBB->lastMIRInsn;
        …
       /* curOffset 是最后一个字节码的地址*/
       curOffset = lastInsn->offset;
       unsigned int targetOffset = curOffset;
     /*最后一个字节码的地址+最后一个字节码的宽度，很显然这是在没有跳转的情况下下一个
       基本块的地址，把这种情况叫做 fallThrough*/
       unsigned int fallThroughOffset = curOffset + lastInsn->width;
       bool isInvoke = false;
       const Method *callee = NULL;
       /*bool findBlockBoundary(…)函数的工作是分析最后一个字节码，如果是 INVOKE
         类指令，就把 callee 函数地址放在 targetOffset，如果是 GOTO，IF_ XX 类指令
         就把目标地址放在 targetOffset */
       findBlockBoundary(desc->method, curBB->lastMIRInsn, curOffset,
                         &targetOffset, &isInvoke, &callee);

       /* Link the taken and fallthrough blocks */
       BasicBlock *searchBB;
       /*查表取出最后一个字节码的属性*/
       int flags = dexGetFlagsFromOpcode(lastInsn->dalvikInsn.opcode);
       if (flags & kInstrInvoke) {
           cUnit.hasInvoke = true;
       }
     /*如果不是函数调用，而且目标地址小于当前地址，那么循环就发生了，bool
       compileLoop(…)专门用来处理循环*/
       if (isInvoke == false &&
           (flags & kInstrCanBranch) != 0 &&
           targetOffset < curOffset &&
           (optHints & JIT_OPT_NO_LOOP) == 0) {
           …
           /*编译循环体*/
           return compileLoop(&cUnit, startOffset, desc, numMaxInsts,
                              info, bailPtr, optHints);
       }
```

```
    /*下面依次扫描所有的基本块，检查它们的起始地址是否是当前块的跳转目标地址，或者
      是紧邻当前块的地址*/
    size_t searchBlockId;
    for (searchBlockId = blockId+1; searchBlockId < blockList->numUsed;
         searchBlockId++) {
        searchBB = (BasicBlock *) dvmGrowableListGetElement(blockList,
searchBlockId);
      /*表明当前块最后一条指令的跳转地址是 searchBB 的起始地址*/
        if (targetOffset == searchBB->startOffset) {
          /* taken 描述两者的跳转/调用关系*/
            curBB->taken = searchBB;
           /*当前块在 searchBB 基本块 predecessors 位图置位 */
            dvmCompilerSetBit(searchBB->predecessors, curBB->id);
        }
        /*表明当前块最后一条指令后面的地址就是 searchBB 的起始地址*/
        if(fallThroughOffset == searchBB->startOffset) {
                /* taken 描述两者的顺序执行关系*/
                curBB->fallThrough = searchBB;
                /*当前块在 searchBB 基本块 predecessors 位图置位 */
          dvmCompilerSetBit(searchBB->predecessors, curBB->id);
                ...
        }
    }
...
}
```

17.8.6　寄存器分配

1. 寄存器分配的步骤

（1）统计当前函数使用的 Dalvik 虚拟机寄存器，参见“Dalvik 寄存器编译模型”一节。

（2）分析每一条 MIR 的操作数，为其分配寄存器（不是实际的 ARM 寄存器，仅为抽象的寄存器），并将该寄存器与 Dalvik 虚拟机寄存器建立对应关系。

（3）在 Dalvik jit 里有个 SSA 概念，它代表一个抽象的寄存器组。SSA 的寄存器组跟 Dalvik 寄存器组一一对应。每个 struct CompilationUnit 结构包含两个数组 int *dalvikToSSAMap;和 GrowableList *ssaToDalvikMap;。前者以 Dalvik 寄存器号做数组索引可以找到对应的 SSA 寄存器号，后者以 SSA 寄存器号做数组索引可以找到对应的 Dalvik 寄存器号。

（4）接下来需要分析每条 Dalvik 指令的寄存器需求，其做法是扫描每一个 BasicBlock，然后针对每一个 BasicBlock 的每一条 MIR 做寄存器使用情况分析。

```
bool dvmCompilerDoSSAConversion(CompilationUnit *cUnit, BasicBlock *bb)
{
```

```
    MIR *mir;
    …
    for (mir = bb->firstMIRInsn; mir; mir = mir->next) {
    /*struct SSARepresentation 记录每条 Dalvik 指令的抽象寄存器需求*/
        mir->ssaRep = (struct SSARepresentation *)
            dvmCompilerNew(sizeof(SSARepresentation), true);
        /*取出该 Dalvik 操作码的属性，每个操作码属性由若干代表目标操作数、源操作数等属
          性位组成，根据属性位可以分析出需要多少寄存器、数据最终是存到哪个寄存器、是否
          需要浮点寄存器*/
        int dfAttributes =
            dvmCompilerDataFlowAttributes[mir->dalvikInsn.opcode];
        /*源操作数寄存器需求计数*/
        int numUses = 0;
       …
/* 宏 DF_HAS_USE 定义如下
 #define DF_HAS_USES            (DF_UA | DF_UB | DF_UC | DF_UA_WIDE | \
                                DF_UB_WIDE | DF_UC_WIDE)
其中每个 DF_UXX 代表一个源操作数寄存器的需求
*/
        if (dfAttributes & DF_HAS_USES) {
            if (dfAttributes & DF_UA) {
            /*如果需要源操作数寄存器，增加一个寄存器需求*/
                numUses++;
            } else if (dfAttributes & DF_UA_WIDE) {
             /*如果需要源操作数寄存器，但类型为 WIDE，增加两个寄存器需求 */
                numUses += 2;
            }
             /*如果需要源操作数可能有多个，依次检查*/
            if (dfAttributes & DF_UB) {
                numUses++;
            } else if (dfAttributes & DF_UB_WIDE) {
                numUses += 2;
            }
           …
        }
       /*给统计出来的源操作数寄存器分配指针数组和浮点判定数组*/
        if (numUses) {
            mir->ssaRep->numUses = numUses;
            mir->ssaRep->uses = (int *)dvmCompilerNew(sizeof(int) * numUses,
                                                      false);
            mir->ssaRep->fpUse = (bool *)dvmCompilerNew(sizeof(bool) *
            numUses,
                                                 false);
        }
/*目的地操作数寄存器计数器*/
```

```
        int numDefs = 0;
/* 宏 DF_HAS_DEFS 定义如下
#define DF_HAS_DEFS             (DF_DA | DF_DA_WIDE)
其中每个 DF_DXX 代表一个目的地操作数寄存器的需求
*/
        if (dfAttributes & DF_HAS_DEFS) {
/*目的地操作数寄存器只可能有一个，无非是普通类型还是 WIDE 类型*/
            numDefs++;
            if (dfAttributes & DF_DA_WIDE) {
                numDefs++;
            }
        }

/*给统计出来的目的地操作数寄存器分配指针数组和浮点判定数组*/
        if (numDefs) {
            mir->ssaRep->numDefs = numDefs;
            mir->ssaRep->defs = (int *)dvmCompilerNew(sizeof(int) * numDefs,
                                                      false);
            mir->ssaRep->fpDef = (bool *)dvmCompilerNew(sizeof(bool) *
            numDefs,
                                                        false);
        }

        DecodedInstruction *dInsn = &mir->dalvikInsn;
/* 再次检查源操作数*/
        if (dfAttributes & DF_HAS_USES) {
            numUses = 0;
            if (dfAttributes & DF_UA) {
/*前面分配的浮点判定数组的用处在这里体现出来了，如果指令的源操作数浮点属性被置位，显然
这是个浮点操作，fpUse []对应为位被置位*/
                mir->ssaRep->fpUse[numUses] = dfAttributes & DF_FP_A;
/*根据 Dalvik 编码里使用的寄存器号，找到 SSA 对应的寄存器号，并在 uses []数组里记住这
个 SSA 寄存器*/
                handleSSAUse(cUnit, mir->ssaRep->uses, dInsn->vA,
                numUses++);
            } else if (dfAttributes & DF_UA_WIDE) {
/*如果源操作是 WIDE，那么源操作数 A 需要使用两个 Dalvik 寄存器，相应的也需要使用两个 SSA
寄存器，处理方法和前者相同，只是这里需要将 dInsn->vA 和 dInsn->vA+1 分别处理一次*/

                mir->ssaRep->fpUse[numUses] = dfAttributes & DF_FP_A;
                handleSSAUse(cUnit, mir->ssaRep->uses, dInsn->vA,
                numUses++);
                mir->ssaRep->fpUse[numUses] = dfAttributes & DF_FP_A;
                handleSSAUse(cUnit, mir->ssaRep->uses, dInsn->vA+1,
                numUses++);
```

```
            }
            /*下面处理源操作数 B、C，方法和源操作数相同*/
            if (dfAttributes & DF_UB) {
              …
            } else if (dfAttributes & DF_UB_WIDE) {
             …
            }
            if (dfAttributes & DF_UC) {
             …
            } else if (dfAttributes & DF_UC_WIDE) {
             …
            }
        }
          /* 处理目的地操作数*/
        if (dfAttributes & DF_HAS_DEFS) {
          /*若为浮点数寄存器，fpUse []对应为位被置位*/
            mir->ssaRep->fpDef[0] = dfAttributes & DF_FP_A;
          /*在 defs []对应项记下是哪个 SSA 寄存器*/
            handleSSADef(cUnit, mir->ssaRep->defs, dInsn->vA, 0);
            if (dfAttributes & DF_DA_WIDE) {
 /*目的地操作也有可能是 WIDE，这里处理第二个寄存器，第一个寄存器前面已经默认处理了*/
                mir->ssaRep->fpDef[1] = dfAttributes & DF_FP_A;
                handleSSADef(cUnit, mir->ssaRep->defs, dInsn->vA+1, 1);
            }
        }
    }
…
}
```

2. ARM 寄存器的使用

```
/*可用的整数寄存器，里面少了 r5 r6，因为 r5 r6 分别被 Dalvik 解释器当 rFP 和 rSELF，
使用这两个不能被自由分配。而 r4 通常会被 EXPORT_PC()保存起来可以被恢复，所以 r4 也可自
由分配，r13 r14 r15 分别是 sp、lr、pc，所以这些寄存器也不能自由分配
*/
static int coreTemps[] = {r0, r1, r2, r3, r4PC, r7, r8, r9, r10, r11, r12};
/*可用的浮点寄存器*/
static int fpTemps[] = {fr16, fr17, fr18, fr19, fr20, fr21, fr22, fr23,
                        fr24, fr25, fr26, fr27, fr28, fr29, fr30, fr31};


void dvmCompilerInitializeRegAlloc(CompilationUnit *cUnit)
{
    …
    /*寄存器池：ARM 寄存器的整体管理结构*/
    RegisterPool *pool = (RegisterPool *)dvmCompilerNew(sizeof(*pool),
```

```
    true);
    cUnit->regPool = pool;
    pool->numCoreTemps = numTemps;
    /*每个寄存器对应一个 struct RegisterInfo，为每个可用整数寄存器分配 struct
RegisterInfo 结构*/
    pool->coreTemps = (RegisterInfo *)
            dvmCompilerNew(numTemps * sizeof(*cUnit->regPool->coreTemps),
            true);
pool->numFPTemps = numFPTemps;
/*每个可用浮点寄存器分配 struct RegisterInfo 结构*/
    pool->FPTemps = (RegisterInfo *)
            dvmCompilerNew(numFPTemps * sizeof(*cUnit->regPool->FPTemps),
            true);
    /*为每个可用寄存器初始化 struct RegisterInfo 结构，主要内容如下：
    编号。每个寄存器都有自己的编号，该编号在 enum NativeRegisterPool 里定义；
    是否被使用。该寄存器是否被分配出去了；
    是否配对使用。本寄存器是否需要跟其他寄存器一起使用
    */
    dvmCompilerInitPool(pool->coreTemps, coreTemps, pool->numCoreTemps);
    dvmCompilerInitPool(pool->FPTemps, fpTemps, pool->numFPTemps);
    pool->nullCheckedRegs =
        dvmCompilerAllocBitVector(cUnit->numSSARegs, false);
}

extern RegLocation dvmCompilerEvalLoc(CompilationUnit *cUnit, RegLocation
loc,
                                      int regClass, bool update)
{
    ...
/*先从已分配出去的 ARM 寄存器里找，是否有对应当前 SSA 寄存器的*/
    loc = dvmCompilerUpdateLoc(cUnit, loc);
...
/*在尚未分配出去的可自由分配 ARM 寄存器里找一个 ARM 寄存器*/
    newReg = dvmCompilerAllocTypedTemp(cUnit, loc.fp, regClass);
    loc.lowReg = newReg;

    if (update) {
        loc.location = kLocPhysReg;
        /*这个新分配出来的 ARM 寄存器对应地记录下 SSA 寄存器号*/
        dvmCompilerMarkLive(cUnit, loc.lowReg, loc.sRegLow);
    }
    return loc;
}
```

17.8.7 LIR

LIR 与机器架构相关，每个 LIR 对应相应机器指令，有自己的操作码和操作数。对于每一个 MIR，生成若干 LIR。LIR 的生成有如下两个步骤：

（1）查找字节码的格式，尽管 Dalvik 字节码有 200 多个，但是根据其操作数的个数和格式可以将其分为 36 大类。通常相同功能的字节码的编码格式都相同，如字节码 OP_IF_EQZ、OP_IF_NEZ、OP_IF_LTZ、OP_IF_GEZ、OP_IF_GTZ、OP_IF_LEZ 都属于 Fmt21t 类型的字节码编码格式。

（2）根据某种格式的字节码，并生成相关的 ArmLIR 并分配对应的 ARM 操作码类型，对于 v7 架构使用的 thumb2 指令集。值得注意的是，并不是一条 MIR 对应一条 LIR，实际上根据不同的 MIR 功能，通常会有多条 LIR 来实现，代码如下：

```
void dvmCompilerMIR2LIR(CompilationUnit *cUnit)
{…
/*分为两个层次，首先依次取出每个基本块，然后针对每个基本块的 MIR 链表生成对应 LIR，对于
ARM 架构 LIR 的实现为 struct ArmLIR*/
  for (BasicBlock *nextBB = bb; nextBB != NULL; nextBB = cUnit->
  nextCodegenBlock) {
           bb = nextBB;
           /visited 标识着该基本块将产生 LIR/
           bb->visited = true;
           cUnit->nextCodegenBlock = NULL;
           for (mir = bb->firstMIRInsn; mir; mir = mir->next) {
           …
              /*取出 Dalvik 字节码*/
               Opcode dalvikOpcode = mir->dalvikInsn.opcode;
               /*分析出该字节码格式*/
               InstructionFormat dalvikFormat =
                   dexGetFormatFromOpcode(dalvikOpcode);
                 …
           /*根据该字节码的类型进行不同的处理*/
                   switch (dalvikFormat) {
                       …
                       case kFmt21h:
                           notHandled = handleFmt21h(cUnit, mir);
                           break;
                       …
                       case kFmt21t:
                       /*OP_IF_XXX 类型字节码，以此例详细分析*/
                           notHandled = handleFmt21t(cUnit, mir, bb,
                                                     labelList);
                           break;
                     …
```

```
                        default:
                            notHandled = true;
                            break;
                    }
                }
            }
...
}
```

下面以 OP_IF_XXX 字节码为例，分析其 LIR 的生成，代码如下：

```
static bool handleFmt21t(CompilationUnit *cUnit, MIR *mir, BasicBlock *bb,
                         ArmLIR *labelList)
{
   /**/
    Opcode dalvikOpcode = mir->dalvikInsn.opcode;
    ArmConditionCode cond;
    /* 通过检查目标地址与当前地址的先后来判定是否 backward 跳转*/
    bool backwardBranch = (bb->taken->startOffset <= mir->offset);
    ...
    RegLocation rlSrc = dvmCompilerGetSrc(cUnit, mir, 0);
    rlSrc = loadValue(cUnit, rlSrc, kCoreReg);

    /*生成比较指令的 ArmLIR，使用的 ARM 操作码为 kThumbCmpRR*/
    opRegImm(cUnit, kOpCmp, rlSrc.lowReg, 0);

    /*跟据 Dalvik 字节码的生成条件码*/
    switch (dalvikOpcode) {
        case OP_IF_EQZ:
        /*条件码位等于*/

            cond = kArmCondEq;
            break;
        case OP_IF_NEZ:
            /*条件码位等于*/
            cond = kArmCondNe;
            break;
        case OP_IF_LTZ:
            /*条件码位小于*/
            cond = kArmCondLt;
            break;
        case OP_IF_GEZ:
             /*条件码位大于*/
            cond = kArmCondGe;
            break;
            ...
```

```
        default:
          …
    }
    /*生成 kThumb2BCond 指令的 ArmLIR，该指令条件码已在前面被检出。这是 taken 分支，跳转
      到另一个 tracerun */
      genConditionalBranch(cUnit, cond, &labelList[bb->taken->id]);
    /* This mostly likely will be optimized away in a later phase */
    /*生成 kThumbBUncond 指令的 ArmLIR，这是 fallThrough 分支，正如注释里指出的一样，这
      条 LIR 可能被优化掉 */
      genUnconditionalBranch(cUnit, &labelList[bb->fallThrough->id]);
      return false;
    }
```

17.8.8 Codecache

Codecache 即为存放机器代码段的 cache，其中包括如下类型的代码。

（1）基本例程。这些代码是 jit 的辅助例程，供 JIT 生成代码使用。其实现在：

```
./dalvik/vm/compiler/template/out/CompilerTemplateAsm-armv7-a-neon.S
```

（2）编译代码。根据 Dalvik 字节码生成的机器代码段。

（3）中间代码。用来进入、连接译代码的代码，通常根据特殊基本块生成。

每一个基本例程对应 intptr_t templateEntryOffsets[]数组的其中一项，代表该基础例程偏移基本例程的起始地址 dvmCompilerTemplateStart 的长度。

```
#define JIT_TEMPLATE(X) templateEntryOffsets[i++] = \
/*例程地址-起始地址*/
    (intptr_t) dvmCompiler_TEMPLATE_##X - (intptr_t) dvmCompiler
    TemplateStart;
/*该文件定义了每个例程名*/
#include "../../../template/armv5te-vfp/TemplateOpList.h"
#undef JIT_TEMPLATE

bool dvmCompilerSetupCodeCache(void)
{   …
    /* 创建 code cache 的虚拟内存 */
    fd    =    ashmem_create_region("dalvik-jit-code-cache",    gDvmJit.
codeCacheSize);
    …
    gDvmJit.codeCache = mmap(NULL, gDvmJit.codeCacheSize,
                             PROT_READ | PROT_WRITE | PROT_EXEC,
                             MAP_PRIVATE , fd, 0);
     …
//把基础例程拷贝到 codecache 最开始处
    memcpy((void *) gDvmJit.codeCache,
```

```
            (void *) dvmCompilerTemplateStart,
            templateSize);
        ...
    }
```

17.9　Dalvik ART

截至本书即将完成之时，Android 又推出了 ART。笔者本来不想在没有读过相关代码之前发表评论，但是 ART 在架构之中的位置如此重要，将会在未来引起 Android 架构惊心动魄的演进，笔者禁不住要将以下内容加入本书。

首先，解释器不是虚拟机全部。虚拟机由进线程管理、内存管理、基础类库等组成。笔者认为解释器应该属于虚拟机进线程管理的一部分。ART 可以理解为一种特殊的解释器，一个将解释器指令 handler 替换掉对应字节码的二进制镜像。所以 ART 并不是替换掉了 Dalvik 虚拟机而是为 Dalvik 提供了一种新的解释器，Dalvik 的进线程管理、内存管理、基础类库还在，寄存器编译模型还在。

ART 带来的负面影响是其代码长度大幅增加，即使能够进行优化，也是字节码长度和字节码 handler 长度的区别。这样导致代码密度增大，存储空间和内存使用增大。

不过存储空间、内存空间是很容易获取的，随着半导体技术的发展，其单位成本都在急剧下降。那么可以期待：改进解释器使其能够像执行 native 函数一样执行 ART 化的类成员函数；把基础类库 ART 化；将某些热点函数编译并保存下来；支持 Java 语法使其在函数定义时能够指出其属性在编译时直接编译为二进制。

也许 Java 的捍卫者会说这样会破坏了 Java 的精髓，但是在手机和嵌入式领域，有几个 Java 的拥趸。

第 18 章　Binder

18.1　Parcel

Parcel 不仅仅是一包数据的体现，Parcel 是布局在从 Java 层到内核层系统对象识别传递机制。对象识别最关键的处理在内核，内核里决定 Parcel 里对象的属性，C++层将 Parcel 里的对象实体化，Java 层将 Parcel 在 Java 层实体化，但是尽管在每一层都有实体对象，这些实体对象只是在各层体现不同，其意义都是相同的。系统中大量的 service 对象和非 service 对象的本地远程机制的建立都是依靠 Parcel 机制来完成。本章着重分析 service 对象的本地及远程生成机制，而在一些实例章节分析非 service 对象的本地及远程建立机制。

另外，尽管内核层是对象识别的主体，为了章节划分方便，本节仅分析 Parcel 的在 C++和 Java 层的机制，内核层机制参见 Binder 驱动一节。

18.1.1　C++层的 Parcel

Parcel 的处理分为压包和解包过程，解包较为复杂，且更能体现 parcel 的意义，为节省篇幅本节仅分析解包。

```
//Parce 解包
status_t unflatten_binder(const sp<ProcessState>& proc,
    const Parcel& in, sp<IBinder>* out)
{
    …
    if (flat) {
          switch (flat->type) {
               case BINDER_TYPE_BINDER:
                     //本地对象，直接取出即可
                     *out = static_cast<IBinder*>(flat->cookie);
                     …
              case BINDER_TYPE_HANDLE:
                   //代理对象的处理见下文
                   *out = proc->getStrongProxyForHandle(flat->handle);
                   …
        }
    }
   …
```

```
}
//代理对象的处理
sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
{
    sp<IBinder> result;
    AutoMutex _l(mLock);
/*每个进程有个存储机构，存放着该进程使用的 Ibinder，其索引通过 Ibinder 的 handle 来实
  现，这里是在存储机构查找 handle 为零的 IBinder，如果不存在也为其分配一个存储项
  handle_entry*/
    handle_entry* e = lookupHandleLocked(handle);

    if (e != NULL) {
            IBinder* b = e->binder;
        if (b == NULL || !e->refs->attemptIncWeak(this)) {
        /*0 号 handle 不存在，为其建立 class BpBinder 对象，并初始化其 mHandle 成
          员变量*/
            b = new BpBinder(handle);
            e->binder = b;
            if (b) e->refs = b->getWeakRefs();
            result = b;
        }
        …
    return result;
}
```

18.1.2　Java 层的 Parcel

同 C++层一样，Java 层也是分析其解包过程。不同于 C++层，Java 层没有显示的解包函数，也不需要这样做。Java 层的解包体现在对包内对象的引用上。

```
//Java 层的 Parcel
public final class Parcel {
  …
  //对象引用，通过该函数获得 Java 层对象
  public final IBinder readStrongBinder() {
          return nativeReadStrongBinder(mNativePtr);
    }
…
}
```

在 JIN 层 nativeReadStrongBinder 函数调用如下函数：

```
static jobject android_os_Parcel_readStrongBinder(JNIEnv* env, jclass
clazz, jint nativePtr)
{
    …
```

```
if (parcel != NULL) {
    /*C++层的 readStrongBinder 其实就是 unflatten_binder，本地对象获得 BBinder，
      代理对象获得 BpBinder*/
        return javaObjectForIBinder(env, parcel->readStrongBinder());
    }
    return NULL;
}

//该函数的作用是通过其获得 val 参数，生成其 Java 层的代理对象
jobject javaObjectForIBinder(JNIEnv* env, const sp<IBinder>& val)
{
    …
    if (val->checkSubclass(&gBinderOffsets)) {
        /*跑到这里说明遇到了一个本地对象，但本地对象不在这里生成，且已经存在，这里直接
          找到本地对象返回即可*/
        jobject object = static_cast<JavaBBinder*>(val.get())->object();
        …
    }
    …
    //别的线程抢先一步
    jobject object = (jobject)val->findObject(&gBinderProxyOffsets);
    if (object != NULL) {
         //既然已经生成 weak 引用之
         jobject   res   =   env->CallObjectMethod(object,   gWeakReference
Offsets.mGet);
        …
    }
    /*真正的对象生成，进入 Java 解释器生成对象。准确的说是生成一个对象，再进入解释器执
      行其构造函数*/
object = env->NewObject(gBinderProxyOffsets.mClass, gBinderProxyOffsets.
mConstructor);
    //成功生成 class BinderProxy 对象
    if (object != NULL) {
          //在 mobject 记下 Java 层的 BpBinder
          …
    }
    //返回 Java 的时候该对象就已经可用
    return object;
}
```

18.2 Binder 驱动

对象识别最关键一点是区分出其所属进程，这是内核层面的工作，Binder 协议的核心

是由 Binder 内核驱动实现的。在 Binder 驱动中，每个进程的对象被组成一棵树，之间存在互相引用。通过解析接受到对象是否在其树中决定是本地对象还是远程对象。

18.2.1　Binder 写

有多种时机如 Binder 对象的引用发生时，有新的线程加入或退出 Binder 协议都会触发 Binder 写。这些情况或者对架构影响不大或者仅仅是维护 Binder 协议本身的动作，本书不做过多展开。下面着重分析当远程代理对象发起对本地对象调用时产生的 Binder 写，这是 Binder 对象机制以及 Android 线程模型至关重要的地方。

```
/*当远程代理对象调用本地对象，或者向 servicemanager 注册对象或者查询对象时，该函数被
  调用，触发命名为 BC_TRANSACTION。这里 struct binder_proc *proc 和 struct
  binder_thread *thread 分别代表发起 BC_TRANSACTION 的进程和线程*/
  int binder_thread_write(struct binder_proc *proc, struct binder_thread
  *thread,
             void __user *buffer, int size, signed long *consumed)
{
    …

    while (ptr < end && thread->return_error == BR_OK) {
      //从用户态获得命令
        if (get_user(cmd, (uint32_t __user *)ptr))
            return -EFAULT;
        …
        switch (cmd) {
        …
       /* BC_TRANSACTION 远程代理的调用动作，BC_REPLY 对应本地对象的函数返回写动
          作，分别是不同进程的先后写*/
        case BC_TRANSACTION:
        case BC_REPLY: {
            struct binder_transaction_data tr;
          //将用户空间的 struct binder_transaction_data 参数取出来
            if (copy_from_user(&tr, ptr, sizeof(tr)))
                return -EFAULT;
            ptr += sizeof(tr);
          //Binder 核心，对象与进程的区分都在这里进行
            binder_transaction(proc, thread, &tr, cmd == BC_REPLY);
            break;
        }
       …
        }
    }
    return 0;
}
```

每个 Binder 本地对象，无论是 C++还是 Java 层面，其在内核都用一个 struct binder_node 唯一对应。每个进程的所有 struct binder_node 组成一棵树。若要用代理对象访问跨进程的本地对象，则代理对象所在进程的底下必先生成其引用结构 struct binder_ref。

```
/*struct binder_transaction_data*tr 为 transaction 交易参数，reply 表示其方向*/
static void binder_transaction(struct binder_proc *proc,
                              struct binder_thread *thread,
                              struct binder_transaction_data *tr, int reply)
{
    …


    if (reply) {
        //不考虑应答方向，理解了调用方向，应答方向自然迎刃而解
    } else {
      /*target 为远程对象，target.handle 为远程对象的句柄*/
        if (tr->target.handle) {
            struct binder_ref *ref;
            //在 struct binder_ref 树中查找远程
            ref = binder_get_ref(proc, tr->target.handle);
            if (ref == NULL) {
                /*在实现调用之前，必先建立其 struct binder_ref，若找不到引用，接
                  下来无法进行*/
            }
            //远程本地对象
            target_node = ref->node;
        } else {
            //若远程对象句柄为 0，这个远程对象就是 BINDER_SERVICE_MANAGER
            target_node = binder_context_mgr_node;
            …
        }
        …
      /*早期 Binder 不限制调用权限，这其实是个安全漏洞，使得系统组件通过 Binder 暴露
        给所有应用，后期的 Android 加入了 Binder 安全机制*/
        if(security_binder_transaction(proc->tsk, target_proc->tsk) < 0) {
            …
        }
        if (!(tr->flags & TF_ONE_WAY) && thread->transaction_stack) {
          /*若要双向交互则要使用 transaction_stack，为简单起见略去这种情况*/
            …
            }
         …
        }
    }
```

```
    …

    /* 在内核里分配 struct binder_transaction*/
    t = kzalloc(sizeof(*t), GFP_KERNEL);
    …
   /*初始化 struct binder_transaction 基本参数*/
    t->sender_euid = proc->tsk->cred->euid;
  //指出 transaction 对应进程
    t->to_proc = target_proc;
    t->to_thread = target_thread;
  // 指出 transaction 对应的函数
    t->code = tr->code;
    t->flags = tr->flags;
    t->priority = task_nice(current);

    …
   //分配 transaction 的 buffer
    t->buffer = binder_alloc_buf(target_proc, tr->data_size,
        tr->offsets_size, !reply && (t->flags & TF_ONE_WAY));
    …
   //用户打包的对象信息地址
   offp = (size_t *)(t->buffer->data + ALIGN(tr->data_size, sizeof(void
*)));
   //复制用户态携带信息
    if  (copy_from_user(t->buffer->data,  tr->data.ptr.buffer,  tr->data_
size)) {
        …
    }
   //复制用户态对象信息
    if (copy_from_user(offp, tr->data.ptr.offsets, tr->offsets_size)) {
        …
    }
    …
    off_end = (void *)offp + tr->offsets_size;
    /*从应用发下来的数据包里依次取出每个对象并进行分析*/
    for (; offp < off_end; offp++) {
        struct flat_binder_object *fp;
        …
      /*fp 为对象描述结构的头指针*/
        fp = (struct flat_binder_object *)(t->buffer->data + *offp);
        switch (fp->type) {
        case BINDER_TYPE_BINDER:
        case BINDER_TYPE_WEAK_BINDER: {
          /*如果该Binder是BINDER_TYPE_BINDER或BINDER_TYPE_WEAK_BINDER类型,
            那么对于当前进程，该 Binder 对象必定是本地对象*/
```

```
        struct binder_ref *ref;
      /*在内核层查找本地对象的对象数据结构 struct binder_node*/
        struct binder_node *node = binder_get_node(proc, fp->binder);
        if (node == NULL) {
         /*内核第一次看到该对象，比如在 addservice 时，生成本地对象的对象数据结
         构 struct binder_node 并加入本地对象树中*/
            node = binder_new_node(proc, fp->binder, fp->cookie);
            …
        }
        …
     //安全检查
        if (security_binder_transfer_binder(proc->tsk, target_proc->
tsk)) {
            return_error = BR_FAILED_REPLY;
            goto err_binder_get_ref_for_node_failed;
        }
      /*既然该对象是当前线程的本地对象，那对于 target_proc 来说就是远程对象，生
        成属于 target_proc 引用结构并加入其引用树。struct binder_ref 代表一个
        Binder 的跨进程引用，一头连着该 Binder 的 node，一头连着对方近的 struct
        binder_proc 结构。而每个进程对另外一个进程里的 Binder 对象的引用的索引
        也是在这里编号的。struct binder_ref 成员变量 desc 随着进程中跨进程引用
        的增加而增加，这个数值就是 BpBinder 的 handle */
        ref = binder_get_ref_for_node(target_proc, node);
        …
     /*如果该对象在当前进程是 BINDER_TYPE_BINDER 或 BINDER_ TYPE_WEAK_
       BINDER 类型，说明该对象是当前进程的本地对象，那么对于目标进程，该 Binder
       对象必定是远程对象*/
        if (fp->type == BINDER_TYPE_BINDER)
            fp->type = BINDER_TYPE_HANDLE;
        else
            fp->type = BINDER_TYPE_WEAK_HANDLE;
     //该对象在目标进程的句柄号
        fp->handle = ref->desc;
        binder_inc_ref(ref, fp->type == BINDER_TYPE_HANDLE,
                    &thread->todo);

        …
    } break;
  /*如果该 Binder 是 BINDER_TYPE_HANDLE 或 BINDER_TYPE_WEAK_HANDLE 类型，
    那么对于当前进程，该 Binder 对象必定是远程对象*/
    case BINDER_TYPE_HANDLE:
    case BINDER_TYPE_WEAK_HANDLE: {
      /*既然是 proxy，必定事前已经建立了引用，用该 proxy 的 handle 去该进程的引
        用树里去查相应的引用结构 struct binder_ref*/
        struct binder_ref *ref = binder_get_ref(proc, fp->handle);
        if (ref == NULL) {
```

```
            /*根据 binder 协议，不会出现这种情况，否则出错*/
            return_error = BR_FAILED_REPLY;
            goto err_binder_get_ref_failed;
        }
      //安全检查
        if (security_binder_transfer_binder(proc->tsk, target_proc->
tsk)) {
            return_error = BR_FAILED_REPLY;
            goto err_binder_get_ref_failed;
        }
    /*根据引用找到对应的 node，再根据 node 找到该 Binder 实现的进程*/
        if (ref->node->proc == target_proc) {
        /*该 node 的实现进程就是本次 transaction 的目标进程，该 Binder 对象在
          目标进程为本地对象，将其类型改为 BINDER_TYPE_BINDER 和 BINDER_
          TYPE_WEAK_BINDER*/
            if (fp->type == BINDER_TYPE_HANDLE)
                fp->type = BINDER_TYPE_BINDER;
            else
                fp->type = BINDER_TYPE_WEAK_BINDER;

         /*struct binder_node 的成员变量 cookie 记载着该 Binder 在用户态的地
           址，返回到用户态直接引用指针即可*/
            fp->binder = ref->node->ptr;
            fp->cookie = ref->node->cookie;
        /*修改引用计数*/
           binder_inc_node(ref->node, fp->type == BINDER_TYPE_BINDER,
           0, NULL);
            …
        } else {
        /*这是最复杂的情况，设当前进程为 A，transaction 的 target 进程为 B，而
          Binder 的本地实现及 Bbinder 在进程 C。某个进程去 servicemanager 查
          找某个 service 就是这种情况*/
            struct binder_ref *new_ref;
        /*在目标进程里寻找该 Binder 的引用结构*/
            new_ref = binder_get_ref_for_node(target_proc, ref->
            node);
            if (new_ref == NULL) {
            /*找不到说明不符合 Binder 协议，系统报错*/
                return_error = BR_FAILED_REPLY;
                goto err_binder_get_ref_for_node_failed;
            }
        /*尽管该对象的 node 节点只有一个，但是引用在不同的进程都是独立的，找到了
          该对象在 target 进程的引用，自然就取得了其 handle 值 */
            fp->handle = new_ref->desc;
            /*修改引用计数*/
            binder_inc_ref(new_ref, fp->type == BINDER_TYPE_HANDLE,
```

```
NULL);
                ...
            }
        } break;

        case BINDER_TYPE_FD: {
          /*File 类型，用来在两个进程中共享一个文件，而共享文件意味着 map 到该 file
            的内存的共享，用户进程与 surfaceflinger 之间的内存共享就是通过这种方式
            实现的*/
            int target_fd;
            struct file *file;
            ...
          //在目标进程的文件描述符表中分配一项
            target_fd = task_get_unused_fd_flags(target_proc, O_CLOEXEC);
            ...
         /*将 file 安装在文件描述符表中，这样把 fd 传上去，用户层就能使用该文件了*/
            task_fd_install(target_proc, target_fd, file);
            trace_binder_transaction_fd(t, fp->handle, target_fd);
            //这个 handle 即为句柄
            fp->handle = target_fd;
        } break;
       ...
        }
    }
    ...
   //目标进程的 Binder 读时会发现一个 BINDER_WORK_TRANSACTION 待处理
    t->work.type = BINDER_WORK_TRANSACTION;
    list_add_tail(&t->work.entry, target_list);
    ...
    return;
}
```

18.2.2 Binder 读

DVM 的线程池线程趴在 Binder 上，等待有内容从内核传递上来。Binder 读要处理 Binder 协议的各种命令，而最关键的远端代理对象发起 BC_TRANSACTION 时，本地对象的 Binder 读的相应动作。

```
/*struct binder_proc *proc 代表本地对象所在进程，struct binder_thread *thread
代表当前线程池线程*/
static int binder_thread_read(struct binder_proc *proc,
                              struct binder_thread *thread,
                              void __user *buffer, int size,
                              signed long *consumed, int non_block)
{
    ...
```

```
while (1) {
    …
  /*从 Binder 写一节可以看出，远端代理对象线程发起 BC_TRANSACTION 时，已将需要本
    地对象所在线程池线程处理的工作挂在了 thread->todo 上*/
    if (!list_empty(&thread->todo))
        w = list_first_entry(&thread->todo, struct binder_work, entry);
   …


    switch (w->type) {
   //对方通过 Binder 写提交了一个 BINDER_WORK_TRANSACTION 工作
    case BINDER_WORK_TRANSACTION: {
        t = container_of(w, struct binder_transaction, work);
    } break;
    …
    }

    …
   //buffer->target_node 为本地待调用对象
    if (t->buffer->target_node) {
        struct binder_node *target_node = t->buffer->target_node;
      /*cookie 即为其 bbinder 指针，上层线程池线程取出该指针直接就可以执行
        onTransact*/
        tr.target.ptr = target_node->ptr;
        tr.cookie =  target_node->cookie;
        t->saved_priority = task_nice(current);
        …
        //指定命令为 BR_TRANSACTION
        cmd = BR_TRANSACTION;
    } else {
        …
    }
   //指定调用功能函数
    tr.code = t->code;
    …

    tr.data_size = t->buffer->data_size;
    tr.offsets_size = t->buffer->offsets_size;
    tr.data.ptr.buffer = (void *)t->buffer->data +
                proc->user_buffer_offset;
    …
   //将命令和 traction 数据包发到用户态*/
    if (put_user(cmd, (uint32_t __user *)ptr))
        return -EFAULT;
    ptr += sizeof(uint32_t);
```

```
        if (copy_to_user(ptr, &tr, sizeof(tr)))
            return -EFAULT;
        …

        break;
    }
    …
    return 0;
}
```

18.3 C++层面

18.3.1 本地与远端对象

根据本地还是远程对象，C++层有着不同的封装形式，本节讨论这些在 C++层面本地与远端对象的封装的实现。

1. 本地封装

（1）本地 Binder。

```
class BBinder : public IBinder
{
public:
     …
     /*interface 描述符*/
     virtual const String16& getInterfaceDescriptor() const;
     …
    /*接受远端 proxy 的调用*/
    virtual status_t   onTransact( uint32_t code,
                                        const Parcel& data,
                                        Parcel* reply,
                                        uint32_t flags = 0);   ...
}
```

（2）本地服务都从 class BnInterface 继承而来，本地服务最重要的是重载函数 virtual status_t onTransact(..);，远端 proxy 的调用都通过该函数进来。

```
template<typename INTERFACE>
class BnInterface : public INTERFACE, public BBinder
{
public:
    virtual sp<IInterface>     queryLocalInterface(const String16&
```

```
        _descriptor);
        …
};
```

2. proxy 封装

```
class BpRefBase : public virtual RefBase
{
protected:
        …
        /*远端 binder 对象，对远端服务对象的调用都是依据 mRemote 进行的*/
       inline  IBinder*      remote()            { return mRemote; }
        …
        IBinder* const          mRemote;
}
```

远端代理需要继承 class BpInterface，代码如下：

```
template<typename INTERFACE>
class BpInterface : public INTERFACE, public BpRefBase
{
public:
            BpInterface(const sp<IBinder>& remote);
            …
};
```

无论是 class BnInterface 还是 class BpInterface 都继承于某个服务的 INTERFACE。所有服务实现自己的 INTERFACE 都必须遵循以下做法：

（1）继承自 class IInterface。

（2）实现 descriptor 和 asInterface 函数。

实际上每个服务的 INTERFACE 在自己的类定义里引用宏。

DECLARE_META_INTERFACE(INTERFACE)声明 descriptor 和 asInterface 函数。而这两个函数的实现是通过宏 IMPLEMENT_META_INTERFACE(INTERFACE, NAME)完成的。这两个宏的定义位于 frameworks/base/include/binder/IInterface.h。

其中 asInterface 函数是关键，进一步分析其实现，代码如下：

```
   android::sp<I##INTERFACE> I##INTERFACE::asInterface(     \
           const android::sp<android::IBinder>& obj)        \
    {                                                       \
       android::sp<I##INTERFACE> intr;                      \
       if (obj != NULL) {                                   \
           intr = static_cast<I##INTERFACE*>(               \
               obj->queryLocalInterface(                    \
                   I##INTERFACE::descriptor).get());        \
           if (intr == NULL) {                              \
```

```
                intr = new Bp##INTERFACE(obj);                \
            }                                                 \
        }                                                     \
        return intr;                                          \
    }
```

这里参数 obj 的类型或者是 class BBinder 或者是 class Bpinder。如果是 class BBinder，则该指针就是 class BnInterface 的指针，其 queryLocalInterface 函数已经被重载，代码如下：

```
template<typename INTERFACE>
inline sp<IInterface> BnInterface<INTERFACE>::queryLocalInterface(
         const String16& _descriptor)
{/*如果就是自己的接口描述符，则返回自己的指针*/
    if (_descriptor == INTERFACE::descriptor) return this;
    return NULL;
}
```

但是如果是 class Bpinder 类型，其 queryLocalInterface 一直没重载还是 class IBinder 的实现，代码如下：

```
sp<IInterface>  IBinder::queryLocalInterface(const String16& descriptor)
{  /*返回空值*/
    return NULL;
}
```

如果返回 NULL，则生成一个 Bp##INTERFACE 类型的对象，这是一个远端 proxy。值得关注的是 Bp##INTERFACE 类型的对象的 mRemote，指向其对应的 BpBinder。

```
///BpBiner 的 transact 接力
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    // Once a binder has died, it will never come back to life.
if (mAlive) {
/*远程调用，mHandle 标识调用指向的对象，code 为调用的功能， data 为调用参数，reply
  为返回结果*/
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }

    return DEAD_OBJECT;
}
```

18.3.2　服务的建立

本节分析 Service 与 ServiceManager 之间的关系，包括 Service 的 ServiceManager 代理的获取，及通过该代理的注册动作。

1．DefaultServiceManager 的获取

所有服务都被 ServiceManager，系统中需要使用某个服务时，首先要向 ServiceManager 申请：

```
sp<IServiceManager> defaultServiceManager()
{
    /*defaultServiceManager 的结果放在 gDefaultServiceManager，如果已经取得
defaultServiceManager 直接返回*/
    if (gDefaultServiceManager != NULL) return gDefaultServiceManager;
    {
      ...
    /*第一次取 defaultServiceManager */
        if (gDefaultServiceManager == NULL) {
     /*asInterface 函数根据 BpBinder 生成 class BpServiceManager 的对象*/
            gDefaultServiceManager = interface_cast<IServiceManager>(
     /*因为 ServiceManager 单独在一个进程实现，所以这里返回的必定是 BpBinder*/
                ProcessState::self()->getContextObject(NULL));
        }
    }
    return gDefaultServiceManager;
}
```

2．ServiceManager 的 BpBinder 对象生成

```
sp<IBinder> ProcessState::getContextObject(const sp<IBinder>& caller)
{  /*查找 handle 号为 0 的 IBinder*/
    return getStrongProxyForHandle(0);
}
```

3．服务注册

服务首先需要使用 class BpServiceManager:: addService 向 serveicemanager 注册自己，代码如下：

```
    virtual status_t addService(const String16& name, const sp<IBinder>&
service)
    {
        Parcel data, reply;
```

```
        data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
        /*服务名*/
        data.writeString16(name);
        /* service 为自己的 BnXXX 对象*/
        data.writeStrongBinder(service);
        /*BpBinder 的 transact 函数*/
        status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data,
&reply);
        return err == NO_ERROR ? reply.readExceptionCode() : err;
    }
```

18.4 Java 层面

本地对象代理对象在 Java 与 C++层的概念完全一样。Java 层的每个本地对象和代理对象在 C++层都有着唯一对应的对象。所有与 Binder 协议底层相关的操作都是通过这些 C++层的对应对象实现的，然后通过 Java 与 JNI 的进出机制与 Java 层对象进行交互。

Java 层本地对象如下：

```
public class Binder implements IBinder {
    …
    //线程池入口
    public static final native void joinThreadPool();
    …
    private native final void init();
    …
    //线程池线程从这里进入 Java 层
    private boolean execTransact(int code, int dataObj, int replyObj,
            int flags) {
        ...
        try {
            //调用 onTransact 函数，功能类通常会重载 onTransact
            res = onTransact(code, data, reply, flags);
        } catch (…) {
            …
        }
        …
    }
}
```

在 JNI 层有一个 class JavaBBinderHolder 类，它维护一个 class JavaBBinder : public BBinder 类型的对象，该对象是 Java 层本地对象在 C++层的对应对象，线程池通过这个对象访问 Java 层。而内核识别对象和引用也是通过该对象。

```
//Java 层代理对象
```

```
final class BinderProxy implements IBinder {
…
//功能类将调用该函数以实现远程调用
public native boolean transact(int code, Parcel data, Parcel reply,
              int flags) throws RemoteException;
…
}
```

首先在 Java 获得 Parcel 以后，使用 readStrongBinder 来获得其中的对象。readStrongBinder 通过 native 函数 nativeReadStrongBinder 在 C++层解包，这样就获得了该对象的 BpBiner。接着 nativeReadStrongBinder 调用 JNI 层的 newobject，创建 Java 层 class BinderProxy 对象。JIN 在创建 Java 层 class BinderProxy 对象是以先前创建的 Bpbinder 做参数，并在 class BinderProxy 对象的 mObject 位置处记下这个 BpBinder。这样 BpBinder 自然与 class BinderProxy 对应起来。

以后每次 class BinderProxy 调用 transact 找到 BpBinder，从而发起 C++层 transact。

18.5　service_manager

首先，service_manager 是一个单独的进程，早期的 Android 版本是用 Java，由于这个地方的交互十分频繁，所以后来改成 C 的。

service_manager 位于 frameworks/base/cmds/servicemanager/service_manager.c 中。它负责管理系统中所有的 service，它调用 int binder_become_context_manager(…)向内核申明自己就是 Servicemanager。

事实上 service_manager 进程本身没有太多逻辑功能，真正的作用在于内核在其底部维护所有 service 的引用，这样在以后的进程查找 service 时，内核就通过 service_manager 进程下的引用树，找到其节点所在的进程。就逻辑实现上 service_manager 可以作为系统线程存在，为什么单独成为一个进程？这里就是答案。

观察 Java 层的 ServiceManager 就可以发现大量的 native 函数，具体分析其实现就可以发现 class ServiceManager 其实指向 service_manager 进程的，Java 层的类注册、查询服务还是通过 service_manager 进程实现的，或者准确地说是 service_manager 进程下的那棵树实现的。

第 19 章　Class

逻辑上来看，Dalvik 虚拟机可以分割为两部分，一部分是内核的延伸，如内存机制延伸为 Java 堆的构建，对象的分配与 GC、进线程机制延伸为以解释器为中心的线程机制。另一部分就是类机制，大量的类库不仅为 Java 进程提供存储、网络通信等基础功能，而且通过类库内部的有机设计，提供了 Java 进程的应用框架。

19.1　系 统 类 库

19.1.1　Inital class

Inital class 是 Java 语言里最基础的类，Dalvik 系统库构建的第一个工作是构建支持 Java 语言基本类型的 Inital class，这些基础类列表如下所示：

（1）Ljava/lang/Class;

（2）Ljava/lang/Boolean;,Ljava/lang/Character;,Ljava/lang/Float;,Ljava/lang/Double;,Ljava/lang/Byte;,Ljava/lang/Short;,Ljava/lang/Integer;,Ljava/lang/Long;

这些类不是从类文件中加载，而是默认集成进 Dalvik 运行时，且在较早的阶段完成。

```
static bool createInitialClasses() {
    …
    //class Class 对象的生成
    ClassObject* clazz = (ClassObject*)
        dvmMalloc(classObjectSize(CLASS_SFIELD_SLOTS), ALLOC_NON_MOVING);
    …
    DVM_OBJECT_INIT(clazz, clazz);
    SET_CLASS_FLAG(clazz, ACC_PUBLIC | ACC_FINAL | CLASS_ISCLASS);
    clazz->descriptor = "Ljava/lang/Class;";
    /*在 struct DvmGlobals gDvm;的 classJavaLangClass 里记录这个代表 class Class
      的 ClassObject*/
    gDvm.classJavaLangClass = clazz;

    //Java 里面 Void Byte 等原始类型的 ClassObject 的生成

    bool ok = true;
/*在 struct DvmGlobals gDvm;的 classJavaLangClass 里记录这个代表 Void 的
ClassObject*/
```

```
ok &= createPrimitiveType(PRIM_VOID,    &gDvm.typeVoid);
/*在 struct DvmGlobals gDvm;的 classJavaLangClass 里记录这个代表 Boolean 的
  ClassObject*/
    ok &= createPrimitiveType(PRIM_BOOLEAN, &gDvm.typeBoolean);
    …
    return ok;
}
```

以上工作只是初始化了 Inital class，但是还并未将其加载到 DVM 类管理基础设施中。这里只是初始化这些 class 并记录在 DVM 全局变量中。真正加载工作在 static bool initClassReferences();和 bool dvmValidateBoxClasses()里完成。

19.1.2　ODEX 文件的加载

尽管 JAR 文件是常见的类库文件，但这是个压缩文件，Davik 真正使用的是经过解压优化过之后的 ODEX 文件，在 Android 系统类库目录 system/framework 下，每个 JAR 类库都对应一个 ODEX 文件。

ODEX 文件的加载有两个方面的动作：首先是文件通过映射方式加载，这种方式的好处是仅在访问时才会启动磁盘操作，且代码段可共享 page cache，参见本书内核部分分析；第二个动作是文件结构的解析，由于 ODEX 文件已经经过优化，其解析主要完成地址重地位即可。

```
//ODEX 文件映射主体
int dvmDexFileOpenFromFd(int fd, DvmDex** ppDvmDex)
{
    …
    //文件映射，建立 File backed 类型的虚拟内存
    if (sysMapFileInShmemWritableReadOnly(fd, &memMap) != 0) {
        …
    }

     //解析，完成地址重定位
    pDexFile = dexFileParse((u1*)memMap.addr, memMap.length, parseFlags);
    …
    //生成 struct DvmDex，Dalvik 运行直接面对的类库文件结构
    pDvmDex = allocateAuxStructures(pDexFile);
   …
}
//Odex 文件的映射
int sysMapFileInShmemWritableReadOnly(int fd, MemMapping* pMap)
{
   …
   //最关键的操作，读写型 MAP_PRIVATE 映射，Android 系统最主要的映射方式
    memPtr  =  mmap(NULL,  length,  PROT_READ  |  PROT_WRITE,  MAP_FILE  |
```

```
MAP_PRIVATE,
          fd, start);
   …
   return 0;
}
```

19.1.3　系统类库

在 Android 运行时，初始化函数 dvmStartup 中调用基础类库的初始化函数 bool dvmClassStartup()。该函数在做完 Ljava/lang/Class;类及 primitive types 类的初始化之后初始基础类库，基础类库被记录在 gDvm.bootClassPathStr 中，gDvm.bootClassPathStr 有以下两种指定方式。

（1）在 dalvik/vm/init.cpp 里，通过提取环境变量取得。

```
static void setCommandLineDefaults()
{   …
   //提取环境变量"BOOTCLASSPATH"
   envStr = getenv("BOOTCLASSPATH");
   ….
}
```

（2）在创建 DVM 虚拟机时通过添加-Xbootclasspath 参数指定。

将-Xbootclasspath 添加到 jint JNI_CreateJavaVM(JavaVM** p_vm, JNIEnv** p_env, void* vm_args) {}的参数数组 void* vm_args。这将在虚拟机创建过程的 int processOptions(int argc, const char* const argv[], bool ignoreUnrecognized)环境中被分析出来，并指定给 gDvm.bootClassPathStr。

系统类库包含如下文件（以“:”做分割）：

```
/system/framework/core.jar:/system/framework/core-junit.jar:/system/fra
mework/bouncycastle.jar:/system/framework/ext.jar:/system/framework/fra
mework.jar:/system/framework/telephony-common.jar:/system/framework/mms
-common.jar:/system/framework/android.policy.jar:/system/framework/serv
ices.jar:/system/framework/apache-xml.jar
```

事实上，以上类库文件都有其对应的 ODEX 优化文件，Dalvik 其实是加载这些 ODEX 文件。基础类库的使用经过两个阶段：第一个阶段是预处理；第二个阶段是真正的加载。而 Android 系统将经常使用的类单独统计出来，在 zygote 初始化时就将这些类预加载到 DVM 管理机构。这样在 fork 新的 Java 进程时这些预加载的类库得到共享。而那些非预加载的类库，Java 进程使用时才进行加载，尽管这些类库的代码因为 page cache 机制得到内存共享，但是这些类库在 DVM 中的管理机构却不是共享内存的。

本节分析第一个阶段——预处理。

```
//基础类库预处理函数
static ClassPathEntry* processClassPath(const char* pathStr, bool
```

```
isBootstrap)
{
    …
    /*为每一个基础类库文件开出一个 ClassPathEntry，形成一个数组：ClassPathEntry*
    cpe*/
    cpe = (ClassPathEntry*) calloc(count+1, sizeof(ClassPathEntry));
    …
    //记录这个数组在 gDvm.bootClassPath 中
    gDvm.bootClassPath = cpe;

    while (cp < end) {
    …
    //处理每一个 FrameWork 类文件，这里只做 map 动作
    prepareCpe(&tmp, isBootstrap)
    …
    }
```

这样基础类库被整理到 gDvm.bootClassPath，以后加载基础类库里的类从这里寻找即可，而对于基础类库本身预处理实现其内存映射，在 zygote 生成新的 Java 进程被集成下来，所以每个 Java 进程中基础类库的映射地址都相同，如下所示。

库文件名映射起始地址映射长度

```
/system/framework/core.odex             45c4b000 350910
/system/framework/core-junit.odex       45fd7000 6bb8
/system/framework/bouncycastle.odex     45fe0000 108720
/system/framework/ext.odex              4610d000 16f550
/system/framework/framework.odex        4629b000 9f11d8
/system/framework/telephony-common.odex46d4f000 121ac0
/system/framework/mms-common.odex       46e89000 1fc58
/system/framework/android.policy.odex   46ead000 9b4d0
/system/framework/services.odex         46f58000 291e60
/system/framework/apache-xml.odex       47218000 150920
```

以上映射地址仅供参考，由于 Android 系统的版本不同，这些地址也会有所不同。以上为 Android 4.0 版本上的映射。

19.1.4　preloaded-classes

基础类库是为了支持基本 Java 语言的运行，对于 Android 系统，除了基础类库以外，应用框架、UI 体系以及大量功能性组件，都是以类库形式出现的，且其中有相当数量的类库被普通的程序大量使用，为了提高运行效率和内存使用效率，需要将这些库预加载进来。这些常用库就称为 preloaded-classes，其预加载过程是 Android 系统启动主要耗时所在。

预加载过程首先要找出需要找出那些类是需要预先加载的，代码如下：

```
InputStream is = ZygoteInit.class.getClassLoader().getResourceAsStream(
```

```
                PRELOADED_CLASSES);
```

这个 Java 语句的背后动作是：

（1）调用 class Class 的 ClassLoader getClassLoader()；函数找到 class ClassLoader。

（2）得到的 class ClassLoader 是 class BootClassLoader。

（3）调用 class BootClassLoader 的 InputStream getResourceAsStream(String resName)；去找 InputStream。其中顺序如下：

① class BootClassLoader 自身实现的 URL getResource(String resName)。

② class BootClassLoader 自身实现的 URL findResource(String name)。

③ class VMClassLoader 的 URL getResource(String name)。

再往下是 class VMClassLoader native 函数实现的。

抽象地看，由于 class BootClassLoade 继承于 class ClassLoader，这里的动作是让 class ClassLoader 去找到需要加载的类列表。该类列表的创建和寻找机制参见上文。

class VMClassLoader 对应的 native 函数如下：

```
static URL getResource(String name) {
/*找到 BootClassPath 有多少个文件路径。所谓 BootClassPath 其实都是 JAR 文件，相当于
  文件路径*/
        int numEntries = getBootClassPathSize();
//针对每一个文件路径（JAR 里）去找里面的 preloaded-classes 文件
        for (int i = 0; i < numEntries; i++) {
            String urlStr = getBootClassPathResource(name, i);
           ….
        }
        return null;
    }
```

class VMClassLoader native 函数 native private static String getBootClassPathResource (String name, int index);的实现代码如下：

```
static void Dalvik_java_lang_VMClassLoader_getBootClassPathResource(
    const u4* args, JValue* pResult)
{
  …
    result = dvmGetBootPathResource(name, idx);
  …
}

StringObject* dvmGetBootPathResource(const char* name, int idx)
{
…
    switch (cpe->kind) {
    case kCpeJar:
        {
```

```
            JarFile* pJarFile = (JarFile*) cpe->ptr;
            //去 JAR 文件包里找 preloaded-classes 文件
            if (dexZipFindEntry(&pJarFile->archive, name) == NULL)
                goto bail;
            //该 JAR 包里有要找的 preloaded-classes 文件，记录该目录
            sprintf(urlBuf, "jar:file://%s!/%s", cpe->fileName, name);
        }
        break;
    …
    }
…
//给 Java 生成返回对象
    urlObj = dvmCreateStringFromCstr(urlBuf);

bail:
    return urlObj;
}
```

而 preloaded-classes 类库明细的定义方式是：在/system/framework/framework.jar 存放着一个名为 preloaded-classes 的文件，该文件记录了 preloadClasses 时需要加载的类。这些类是经过统计工具得出的最常用的类，原始文件就放在 Android 源码的 frameworks/base/preloaded-classes 路径下，编译时被打包进 framework.jar。

对于 Android 4.0，preloaded-classes 文件列举出的类数目达到 2200 多个，对于 Android 2.3，该文件列举出的类数目是 1800 多个。由此可以理解为什么最初的 Android 在 400M 的 ARM9 跑得都很顺畅，而 5 年后，这个级别的处理器跑起来 Android 4.0 却相当费力。

19.2　类　加　载

Dalvik 虚拟机在运行时需要频繁的执行生成对象、调用函数等操作。而所有这些动作的基础都依赖于类。所以虚拟机在执行与某个类相关的动作之前要充分了解类的所有信息，这个过程就是类加载。

19.2.1　类加载框架

类加载的过程如下：

（1）指定或从系统类库里找到待加载类所在的文件，对应于 struct DvmDex，然后再在该文件中解析出该类类定义结构 struct DexClassDef，这相当于类头。

（2）从类文件中加载该类，这一步生成该类所对应 Dalvik 管理结构 struct ClassObject。

（3）类链接，加载该类对应的父类、建立 vtable 等操作。

```
//Dalvik 类加载的框架函数
```

```
static ClassObject* findClassNoInit(const char* descriptor, Object* loader,
    DvmDex* pDvmDex)
{
   …
   /*Dalvik 将加载的类都加入在其 hash 表 gDvm.loadedClasses 中，这是一个缓存机制，
     先在其中查找该类是否存在*/
    clazz = dvmLookupClass(descriptor, loader, true);
    …
     //hash 表中没有该类，下面执行加载动作
    if (clazz == NULL) {
        …
        if (pDvmDex == NULL) {
            //未指定类的加载目录，在系统类库里找
            pDvmDex = searchBootPathForClass(descriptor, &pClassDef);
        } else {
            //指定了该类所在文件，则直接在指定文件中找
            pClassDef = dexFindClass(pDvmDex->pDexFile, descriptor);
        }

        if (pDvmDex == NULL || pClassDef == NULL) {
            if (gDvm.noClassDefFoundErrorObj != NULL) {
                /* 找不到该类的定义，类未定义异常，交给应用程序去处理*/
                dvmSetException(self, gDvm.noClassDefFoundErrorObj);
            } else {
                /* 类定义异常，可能是文件错误，直接抛出异常*/
                dvmThrowNoClassDefFoundError(descriptor);
            }
            goto bail;
        }

        /* 类加载，这里产生该类对应的 struct ClassObject*/
        clazz = loadClassFromDex(pDvmDex, pClassDef, loader);
        …
        /*接下来其他线程有可能获得该 struct ClassObject，但还没完成工作，所以将其锁
          起来*/
        dvmLockObject(self, (Object*) clazz);
        //记录下类的初始化线程号
        clazz->initThreadId = self->threadId;

        /* 将加载的类加入类的 cache 哈希表中*/
        if (!dvmAddClassToHash(clazz)) {
            /*有另外的线程抢先一步加载该类，类似于内核里经常发生的情况*/
            clazz->initThreadId = 0;
            …
            //释放掉不必要的 clazz
```

```
        dvmFreeClassInnards(clazz);
        …
        /* 取现成的 struct ClassObject 即可 */
        clazz = dvmLookupClass(descriptor, loader, true);
        goto got_class;
    }
    …
    //类链接，参见下文
    if (!dvmLinkClass(clazz)) {
        …
        //链接失败的情况，释放掉该类
        goto bail;
    }
    …
    //解锁该 struct ClassObject，其他的线程也可以用了
    dvmUnlockObject(self, (Object*) clazz);
    …
}
```

19.2.2　类加载

类加载并不是把类从文件中读进内存，而是生成其控制结构 struct ClassObject，并解析类相关成员变量、函数等信息。

```
//类加载主题函数
static ClassObject* loadClassFromDex0(…)
{
            …
            //为该类分配 struct ClassObject 结构
        newClass = (ClassObject*) dvmMalloc(size, ALLOC_NON_ MOVING);
…

    //设置该类的 classloader
    SET_CLASS_FLAG(newClass, pClassDef->accessFlags);
    dvmSetFieldObject((Object *)newClass,
                        OFFSETOF_MEMBER(ClassObject, classLoader),
                        (Object *)classLoader);
    //记录该类所在文件 struct DvmDex
    newClass->pDvmDex = pDvmDex;
    …
    //记录该类的父类，在类链接时需要解析其父类
    newClass->super = (ClassObject*) pClassDef->superclassIdx;
    …
/*在解析完类接口、类静态域、类实例域之后解析类函数，DirectMethod 指的是 Java 里的
```

```
static、private、<init>函数*/
if (pHeader->directMethodsSize != 0) {
      //取出 DirectMethod 类型函数的个数
       int count = (int) pHeader->directMethodsSize;
       …
       newClass->directMethodCount = count;
       //为 DirectMethod 类型函数分配数组空间
       newClass->directMethods = (Method*) dvmLinearAlloc(classLoader,
              count * sizeof(Method));
       /*接下来逐个读取每一个函数，值得关注的是，这里的读取并不是真正的读取，只是文件
         位置的解析，该函数的 Dalvik bytecode 二进制文件是被 MAP 到 DVM 进程里，而映
         射动作也早已完成，参见相关章节。这种设计充分利用了 Linux 内核虚拟内存体系最核
         心最有效的机制（其实不只是 Linux，应该说是整个 Unix 家族，包括 Windows）。这
         样就可以在多个 DVM 进程共享一份内存拷贝，而且在物理内存紧张且该段代码不被经常
         调用时，又可以释放掉那块唯一的物理内存。而在被访问时又被 Cache 进 Linux 的 Page
         Cache，被系统所有需要的进程访问。这是 Dalvik 虚拟机设计的巧妙之处，如同它的
         进线程管理，与 Linux 内核真正融为一体*/
       for (i = 0; i < count; i++) {
            dexReadClassDataMethod(&pEncodedData, &method, &lastIndex);
            loadMethodFromDex(newClass,        &method,        &newClass->
directMethods[i]);
           …
       }
       …
   }
   /*VirtualMethod 函数的加载，所谓 VirtualMethod 指的是 Java 的普通函数。而这类函
     数的调用，要通过 Vtable 进行，一个 class 的 vtable 的建立要在 link 阶段进行*/
if (pHeader->virtualMethodsSize != 0) {
      //为 VirtualMethod 类型函数的个数
       int count = (int) pHeader->virtualMethodsSize;
       …
       //为 VirtualMethod 类型函数分配数组空间
       newClass->virtualMethods = (Method*) dvmLinearAlloc(classLoader,
                 count * sizeof(Method));
      /*依次取出 VirtualMethod 类型函数的 struct DexMethodId 结构，并初始化其基本
        信息（如调用规则等），代码映射机制与 DirectMethod 类型函数相同，参见上文*/
       for (i = 0; i < count; i++) {
            dexReadClassDataMethod(&pEncodedData, &method, &lastIndex);
            loadMethodFromDex(newClass,     &method,     &newClass->virtual
Methods[i]);
           …
           }
       }
```

```
        …
    }

    …
    return newClass;
}
```

19.3　对象实体生成

Dalvik 中对象的生成步骤是，首先加载待生成对象所对应的 class 到内存。Dalvik 首先检测该 class 是否被加载到内存，否者需要将该 class 做加载动作，参见类加载部分。接着在 Java 进程 heap 里根据该类描述中指出的类对象所占用内存大小为该对象分配内存，从而形成对象实体。

对象生成的典型方式是通过 Dalvik 的 new 指令，解释器遇到对象生成指令 new 时的 handler 如下（以 C 解释器为例）：

```
HANDLE_OPCODE(OP_NEW_INSTANCE /*vAA, class@BBBB*/)
    {   …
        //取出 Dalvik 指令里指定的待生成对象的类索引
        ref = FETCH(1);
        …
        /*每个 Dalvik 在 DEX 文件控制结构里记录了已经加载到内存中的 class，这里首先检
          查该 class 是否已经被加载了*/
        clazz = dvmDexGetResolvedClass(methodClassDex, ref);
        if (clazz == NULL) {
            //该 class 没有被加载，这里需要先加载该 class
            clazz = dvmResolveClass(curMethod->clazz, ref, false);
            if (clazz == NULL)
                GOTO_exceptionThrown();
        }
        /*隐含在这里机制是，基础类库、常用类库被大多数 Java 进程使用，都事先加载了，在
          新的 dalvik java 进程生成时 fork 了这些类库以及类库控制结构的虚拟内存空间，
          以后新的 dalvik java 进程只是以读操作访问这里类库以及类库控制结构，物理内存
          不用另外分配，共享已经加载的类即可*/
          …
        //在 dalvik java 进程的 heap 堆里分配该对象
        newObj = dvmAllocObject(clazz, ALLOC_DONT_TRACK);
        …
    }
    FINISH(2);
OP_END
```

```
//对象生成第二步，为对象实体分配虚拟内存
Object* dvmAllocObject(ClassObject* clazz, int flags)
{
    Object* newObj;
    /* 基于 bionic 库的 malloc 算法在 java heap 上分配对象。clazz->objectSize 指出
       了该 class 对象所占内存大小*/
    newObj = (Object*)dvmMalloc(clazz->objectSize, flags);
    if (newObj != NULL) {
        /* 将该对象的 ClassObject*    clazz;指针指向该 class 对应的 struct
           ClassObject 结构*/
        DVM_OBJECT_INIT(newObj, clazz);
        …
    }
    return newObj;
}
```

第 20 章　Android 应用框架

将完全符合 Linux 进程模型的 Android 应用程序抽象成 Android APP 开发者角度看到的 Android Java 编程模型，这就是应用框架所起到的作用，本章探讨 Android API、应用框架、Linux 进程模型之间的关系。

Android 的应用框架的特点如下：

（1）进程中有大量本地对象和与之对应的代理对象（位于另一个进程中）。在 C++层面的 status_t IPCThreadState::executeCommand(int32_t cmd)执行 case BR_TRANSACTION:时，本地对象的 status_t BBinder::transact 继而调用 onTransact 函数。这是线程池中的线程，可以跑到 Java 层。DVM 中（无论普通进程还是系统进程）的 Java 远程对象对本地对象的调用都是这种方式实现的。

（2）以 MainLooper 为主体的主线程跑 class ActivityThread 的代码，不停地接受来自系统组件消息并处理之，这演进自 zygote fork 出来线程。

（3）Android UI 体系实现的主体是在应用进程本身，Android UI 系统组件 class WindowManagerService 仅提供事件分发、窗口管理机制等基本控制。应用进程获得一块称之为 surface 的 bitmap。应用所有 UI 的渲染操作都是由运行应用本身之内 UI 类库完成。3D 的实现也是 UI 类库的一部分，只是其实现可由硬件完成。

（4）应用框架的内存、进程管理、文件操作、网络操作、访问控制等都符合 Linux 传统模型，只是 Java 层的编程方式不同而已。

（5）系统组件作为单独的 DVM 进程运行。从内核角度看，这些系统进程与普通线程无异。只是某些访问频繁的线程如 surface flinger 的优先级不同。

20.1　线程池线程

线程池线程是 Android 对象与代理机制的运行实体，这些线程由系统生成，并由系统控制运行，将 Binder 驱动连接到 Java 层是理解对象与代理的调用机制关键。

20.1.1　C++层

在用户层的最底部，线程池线程作为 deamon，检测内核 Binder 驱动并实现 Binder 协议，若有远程对象调用到来，线程池线程执行该函数：

```
//Binder 协议在用户层的最底层实现，仅分析 BR_TRANSACTION 情况
status_t IPCThreadState::executeCommand(int32_t cmd)
{
```

```
    …
    case BR_TRANSACTION:
        {
            binder_transaction_data tr;
            //mIn 自内核而来，将内容考入 tr
            result = mIn.read(&tr, sizeof(tr));
            …
            //根据 tr 初始化 Parcel buffer，这里面携带着参数信息
            Parcel buffer;
            buffer.ipcSetDataReference(
                reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
                tr.data_size,
                reinterpret_cast<const size_t*>(tr.data.ptr.offsets),
                tr.offsets_size/sizeof(size_t), freeBuffer, this);

            const pid_t origPid = mCallingPid;
            const uid_t origUid = mCallingUid;
            //记录调用者 PID UID
            mCallingPid = tr.sender_pid;
            mCallingUid = tr.sender_euid;

            …
            //replay 存放返回给对方的结果
            Parcel reply;
            …
            /*tr.cookie 指出了本地对象的指针，这是内核识别设置的，参考 Binder 驱动*/
            if (tr.target.ptr) {
                sp<BBinder> b((BBinder*)tr.cookie);
                /*本地对象的调用，导致 onTransact 的调用，无论 C++还是 Java 的本地对象
                  都会重载 onTransact，以实现具体功能*/
                const status_t error = b->transact(tr.code, buffer, &reply,
tr.flags);
               …

            } else {
                …
            }


            if ((tr.flags & TF_ONE_WAY) == 0) {
                //给对方返回结果
                sendReply(reply, 0);
            } else {
                …
            }
```

```
            …
        }
        break;
    …
    }
```

以 private class ApplicationThread extends ApplicationThreadNative 为例，该 class 继承自 class Binder 并重载了 public boolean onTransact(…)函数。在远端进程的 class ActivityManagerService 通过 class ApplicationThreadProxy 发起 bindApplication 调用后。应用进程的线程池线程被唤醒，进而导致 class ApplicationThreadNative 的 onTransact 函数被调用，接着将 bindApplication 挂入 class ActivityThread 的消息队列即返回。

可见 onTransact 函数不是事件处理的主体，其主要用来做消息通知，具体的事件处理还是要在主线程里完成。

20.1.2　Java 层

Java 层本地 Binder 在 C++层对应于 class JavaBBinder : public BBinder，onTransact 是 class JavaBBinder 的核心函数。

```
//自 status_t IPCThreadState::executeCommand(int32_t cmd)而来
virtual status_t onTransact(
        uint32_t code, const Parcel& data, Parcel*reply, uint32_t flags = 0)
    {
        …
         //调用 Java 层 class Binder 的 private boolean execTransact(…)函数
        jboolean res = env->CallBooleanMethod(mObject, gBinderOffsets.
mExecTransact,
            code, (int32_t)&data, (int32_t)reply, flags);
         …
    }
```

Java 层本地 Binder 的 private boolean execTransact(…)函数，至此，线程池的线程跑到了 Java 层，并成为 Java 层本地功能对象命令分发的实体线程，这里是 Android 应用框架接驳 Binder 机制关键所在。除了 zygote 的 fork 机制、activitythread 的 mainlooper 机制，该机制与 Binder 一起成为 Android 架构实现中的又一重要机制。

```
private boolean execTransact(int code, int dataObj, int replyObj,
            int flags) {
        Parcel data = Parcel.obtain(dataObj);
        Parcel reply = Parcel.obtain(replyObj);
        …
        try {
            /*对 onTransact 函数的调用，Java 层本地功能对象对此函数重载，已完成自己的
```

```
命令分发*/
            res = onTransact(code, data, reply, flags);
        } catch (…) {
            …
        }
        …
    }
```

20.2　系统侧 Activity 与 Service 的生成控制

Activity 或 Service 的生成有两方面的工作，在系统侧控制 Activity 或 Service 在新的进程生成还是在原有进程中加载新的 Activity 或 Service。前者意味着一个新的 class ActivityThread 运行单元的创建，后者则仅仅意味着新的类被加载和运行。

```
//系统侧生成 Activity
private final void startSpecificActivityLocked(ActivityRecord r,
                  boolean andResume, boolean checkConfig) {
             //从记录里查找对应应用记录
             ProcessRecord     app     =     mService.getProcessRecordLocked
(r.processName,
                   r.info.applicationInfo.uid);
         …
        //如果系统中存在对应应用且有执行线程
        if (app != null && app.thread != null) {
            try {
                app.addPackage(r.info.packageName);
                /*远程交互其对应的 activitythread，调用其 scheduleLaunchActivity
                */
                realStartActivityLocked(r, app, andResume, checkConfig);
                return;
            }
            …
        }
        /*没有对应的应用，这里将生成新的进程，并在其中生成 Activitythread，从而导致
          bindapplicatin 协议的执行*/
        mService.startProcessLocked(r.processName, r.info.applicationInfo,
true, 0,
                "activity", r.intent.getComponent(), false, false);
    }

//再看 service 的生成控制
    private final String bringUpServiceLocked(ServiceRecord r,
            int intentFlags, boolean whileRestarting) {
```

```
        ...
        /*检查该服务是否需要隔离运行，意味着该 service 可以与其他 activity、service
          在同一进程里运行*/
        final boolean isolated = (r.serviceInfo.flags&ServiceInfo. FLAG_
        ISOLATED_PROCESS) != 0;
       ...
        //如果不需要则隔离运行
        if (!isolated) {
            //在系统里寻找包含该 service 的已运行进程
            app = mAm.getProcessRecordLocked(procName, r.appInfo.uid);
            ...
            //找到了对应进程
            if (app != null && app.thread != null) {
                try {
                    app.addPackage(r.appInfo.packageName);
                    /*导致远程 activitythread 的 scheduleCreateService 被调用*/
                    realStartServiceLocked(r, app);
                    return null;
                }
                ...
             }
         }
        ...
        //该服务需要单独在一个进程里运行，或者没有对应的线程
        if (app == null) {
           //生成新进程及 activitythread，并绑定该 service
           if  ((app=mAm.startProcessLocked(procName,  r.appInfo,  true,
intentFlags,
                        "service", r.name, false, isolated)) == null) {
              ...
           }
         ...
        }
    }
```

生成新进程的方式即通过 socket 与 zygote 进程间交互，再通过其 fork 机制生成新的进程。这也是为什么 systemserver 进程里塞满了 ActivityManagerService、PackageManagerService、WindowManagerService 等一大堆服务，而 zygote 里却空空如也。这是因为服务组件之间交互是相当频繁的，把服务都塞在 systemserver 进程里避免了进程间通信，为了其高性能将服务分配到不同的线程即可。而 zygote 是用来 fork 应用进程的，自然里面不能塞服务，不然每个进程 fork 时都把服务带进来，岂不乱了套。

另外，本节描述的是 activity 和 service 生成时系统侧的控制机制，接下来如何加载

activity 和 service 还是由新进程里的 class activitythread 来完成的。

20.3　class ActivityThread

class ActivityThread 类的 main 是 DVM 主线程执行的主体。class ActivityThread 的实现决定了 Android 进程的基本线程模型，如图 20-1 所示。

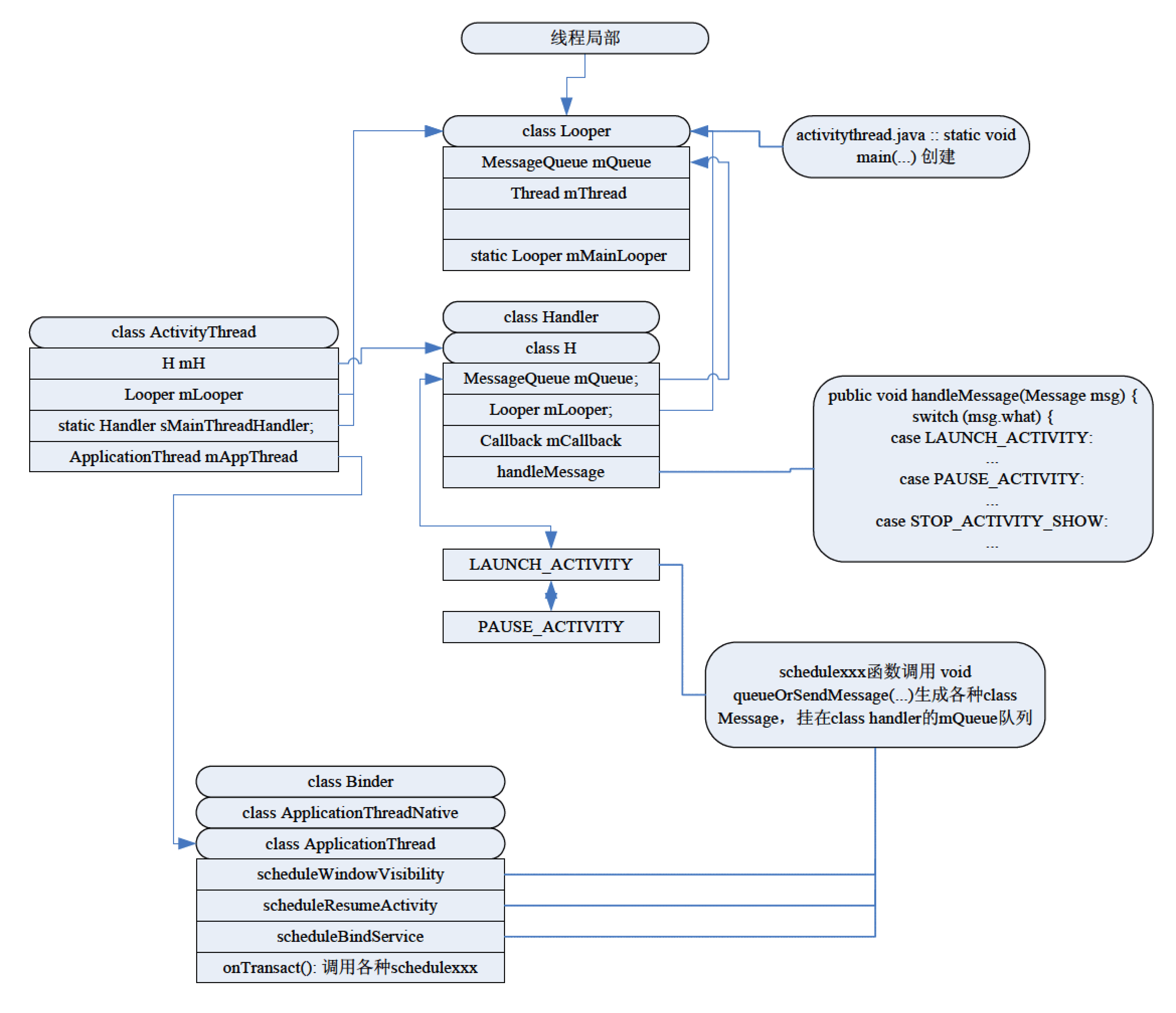


图 20-1　基本线程模型

20.3.1　MainLooper

Looper 机制不是主线程独有的，但是 mainlooper 确是系统中唯一的，它与主线程绑定，是主线程任务分发的中心。

```
public class Looper {
    /*该 looper 所在线程的线程局部存储，在该线程中通过 sThreadLocal 可以访问到该
```

```
looper*/
static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<Looper>();
//一个进程只能有一个 mainlooper、static
    private static Looper sMainLooper;  // guarded by Looper.class
    //looper 的消息队列
    final MessageQueue mQueue;
…
}
```

Looper 使用的第一步是与线程对应起来，代码如下：

```
private static void prepare(boolean quitAllowed) {
           //检查线程局部存储是否已经有了 looper
           if (sThreadLocal.get() != null) {
                 throw new RuntimeException("Only one Looper may be created
per thread");
        }
        //新建一个 looper，并存放在当前线程的局部存储中
        sThreadLocal.set(new Looper(quitAllowed));
}
```

Mainlooper 的构建，代码如下：

```
//looper 的主体
public static void prepareMainLooper() {
          //false 表示主线程不能退出
          prepare(false);
          …
          //记录下 static 的 sMainLooper
           sMainLooper = myLooper();
        }
    }

//looper 主体函数
public static void loop() {
           …
           for (;;) {
                 //不停地从自己的消息队列中取出 message
                 Message msg = queue.next(); // might block
                …
                /*分发该消息。该消息的 target 已经指出了其分发的 handler*/
                 msg.target.dispatchMessage(msg);
                …
         }
}
```

由此可以分析出主线程的工作模式，线程池线程携带来自系统进程的命令调用 class ApplicationThread 的 public boolean onTransact(…)，并将命令以消息队列的形式挂载到主线程的消息队列，指定其 handler 为 class ActivityThread 的私有类 private class H extends

Handler，然后线程池线程返回。

接着主线程 looper 取出消息队列，并执行其分发函数。即为 private class H extends Handler 的 public void dispatchMessage(Message msg)函数。

进一步分析 private class H extends Handler，它就像是一个 DVM 系统管理的门户，诸如 LAUNCH_ACTIVITY、SHOW_WINDOW、BIND_SERVICE、BIND_APPLICATION 等来自系统组件的命令都由这里分发。

20.3.2　activity 与 service 的加载

系统侧决定在当前进程中加载一个新的 activity 时，class Activitythread 的 private void handleLaunchActivity(…)将导致具体执行函数 private Activity performLaunchActivity(…)被调用，代码如下：

```
/*activity 生成通过用户侧与系统之间一套协议来完成，网上已有很丰富的描述资料，这里不介
 绍，本书侧重介绍 Android 进程模型与 activity、service 之间的关系，这里只分析其加载*/
 private Activity performLaunchActivity(ActivityClientRecord r, Intent
 customIntent) {
        …
        try {
           //找到当前应用所在 package 对应的类加载器
            java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
            /*这里其实就是调用类加载器加载该 activity*/
            activity = mInstrumentation.newActivity(
                          cl, component.getClassName(), r.intent);
            …
        }
        …
        return activity;
    }
```

系统侧加载服务调用导致 class Activitythread 的 private void handleCreateService(…)，人们也仅关心其 service 的类加载，代码如下：

```
private void handleCreateService(CreateServiceData data) {
        …
        try {
             //找到该 service 所在 package 对应的类加载器
             java.lang.ClassLoader cl = packageInfo.getClassLoader();
             //加载
            service = (Service) cl.loadClass(data.info.name).newInstance();
        }
        …
}
```

尽管 activity 和 service 创建涉及与系统侧之间复杂绑定协议和 activitythread 的大量管理结构，但是从其类加载机制可以清楚地看出 activitythread 的容器功能，无论是什么类型的 activity 和 service，都不过是 activitythread 里的一个组件，其加载机制决定了 activity、service 与 activitythread 进程的动态关系。

第 21 章　Android UI 体系

Android UI 体系由系统侧和应用侧组成，系统侧由上下两部分组成，上层部分即 class WindowManagerService，主要负责消息的分发、窗口栈的调整，下层部分即 class SurfaceFlinger 负责叠加计算每个应用对应的 surface，并送给底层的 Framebuffer 驱动。

应用侧由 View 树组成，接受服务侧上层 class WindowManagerService 的消息，并进行渲染操作。其渲染结果直接体现在应用侧的 surface 中。3D 硬件加速以 Java 接口的形式暴露给 View 体系，并工作在应用侧。

本章整理自较早版本笔记，代码版本为 Android 1.X，尽管与 Android 4.x 相比有代码滞后的问题，不过架构方面基本相同。其实这也是 Android 的特点之一，由于设计合理，Android 架构一直没有大的变化。这也是 OS 技术层面成功的关键。其实 Android 问世之初很多基本功能并没有完全实现，很多函数甚至是空的。但是由于其合理且弹性的架构设计，版本演进时不需要架构级的改动，这样没有来得及实现的功能逐步加上，系统瓶颈的地方用 C 或汇编替换掉……，每一个版本的工作都没浪费，成为以后版本更新的基础，Android 才能稳步成长到现在。反过来，再看一些技术上失败的 OS 项目，最重要的原因是架构不清晰或者没有弹性，往往动辄数百人的投入，尽管完成了不少局部功能，但是却形成不了合力，架构一改再改，新的功能搭不上去，很多版本被废弃掉，完成不了积累就无法成长，最终导致技术上失败。

21.1　窗口体系的生成

首先 class ActivityManagerService 在启动 Activity 时如果发现 SHOW_APP_STARTING_ICON，则导致 class WindowManagerService 的 public void setAppStartingWindow(…)被调用，接着导致 mPolicy.addStartingWindow(…)被调用，导致 class PhoneWindowManager 的 View addStartingWindow(…)被调用，如图 21-1 所示。

应用程序的 View 框架则是在 class PhoneWindow 的 ViewGroup generateLayout (DecorView decor)函数完成的：

View in = mLayoutInflater.inflate(layoutResource, null);

其中 layoutResource 的 ID 指向 APPS/Common/res/layout/scree_title.xml 中的资源文件。

而应用程序自身定义的 View 树则由 void setContentView(…)生成（该函数由 OnCreate() 调用），其中内容是应用指定的 View 或自身的 layout.xml。

分析 Android 类库可以发现，整个 UI 体系就是一个集成树，如 class GridView、class AbsListView、class AdapterView、class ViewGroup、class View，呈现依次继承关系。

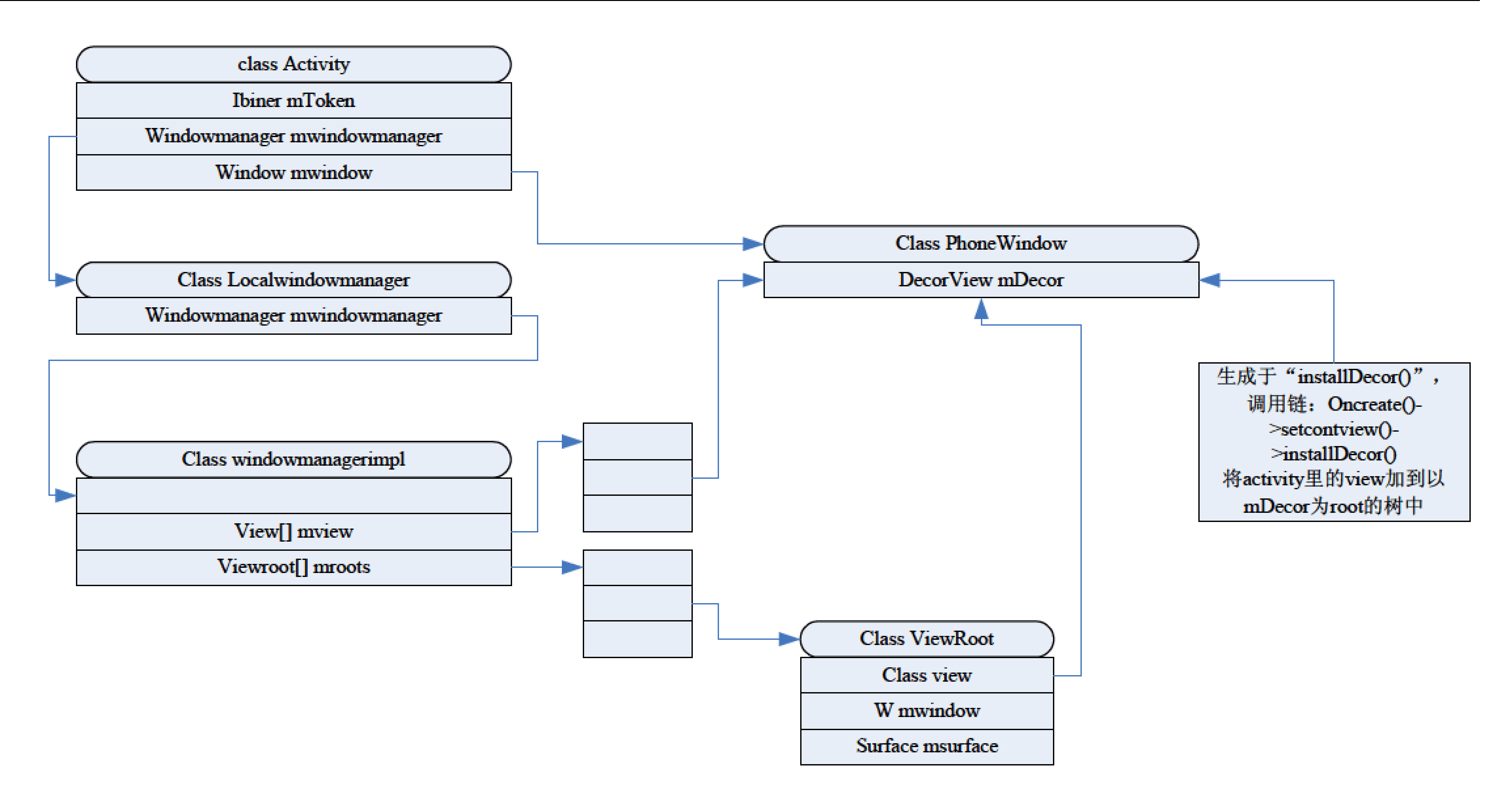


图 21-1　调用过程示意图

class View 是 View 的祖先，它抽象出了一个 View 应该具有的 onlayout、OnDraw、dispatchDraw、dispatchTouchEvnent、OnTouchEvnent 等虚函数。这样不同类型的功能类通过重载这些函数来满足自身需求。

而一个 View 中其他 View 都作为其孩子存在，所以当某个事件发生时，通过对孩子的对应函数调用就能完成渲染和事件响应操作。这种面向对象的思想是 UI 体系发展的方向，不仅解决了传统窗口体系窗口管理器的瓶颈问题，如布局指令的由传统窗口管理器做的工作也可以由 View 体系来完成。而且目前严重依赖 3D 硬件加速的 UI 体系的情况也从这个体系受益匪浅。事实上这个 UI 架构似乎不是 Android 上独有的，不仅 IOS 的 UI 渲染架构与此也非常类似，连传统 X 也似乎要被以 Surface 为中心的 wayland 取代。

21.2　ViewRoot 与 Surface

ViewRoot 与 View 类没有任何集成关系，ViewRoot 是应用 UI 体系与系统侧 WindowManagerService 交互的接口，通过 ViewRoot、WindowManagerService 将输入事件、窗口事件送到 Android 应用，然后再通过 Android View 体系对事件的响应机制达到接受屏幕点击坐标、重新布局、重画 VIEW 等动作。

ViewRoot 的构建时机是在 Activity 完成 View 创建之后，handleResumeActivity()函数被触发，代码如下：

```
final void handleResumeActivity(…) {
…
wm.addView(decor, l);
…
}
```

从而导致 class WindowManagerImpl 的 public void addView(…)被调用，接着触发调用链，如图 21-2 所示。

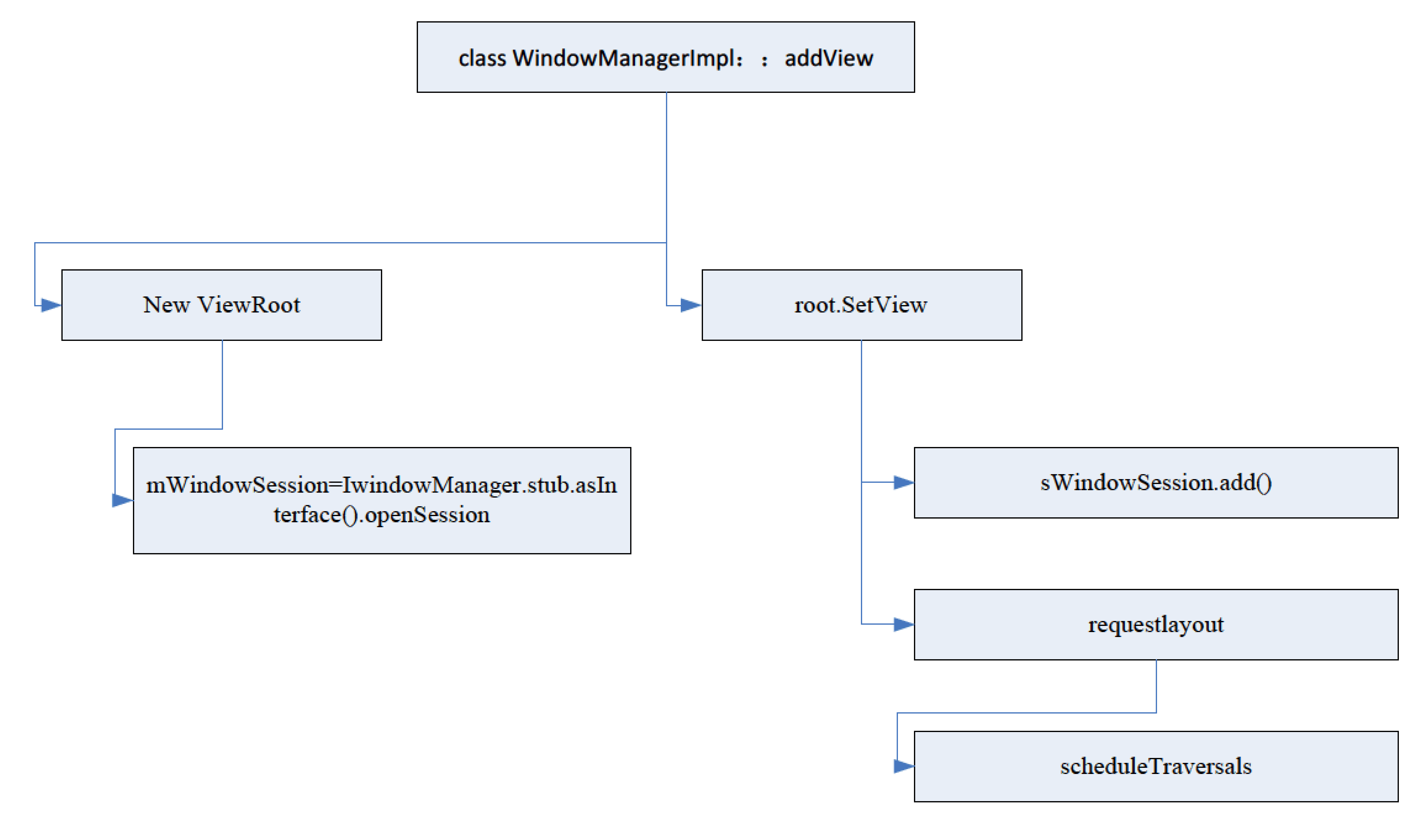


图 21-2　调用链

Surface 是应用进程渲染的最终目的地，本质上是一块位图，且跨进程共享。其共享方式是通过 Binder 传递文件并在文件映射的方式中进行的。Surface 的生成过程如下：

（1）ViewRoot 第一次执行 PerformTraversals 或者 resize 时，导致 sWindowSession.relayout()被调用。

（2）WindowManagerService 创建 Surface。

（3）ViewRoot 将 WindowManagerService 创建的 Surface 记录在自己的 mSurface 中。

（4）在 ViewRoot 发起 Draw 之前调用 Surface.lockcanvas，得到 canvas。

（5）Surface.lockcanvas 实际上就是根据底层支持的 surface 格式创建一个位图。

（6）ViewRoot 把这个 canvas 传给 View 体系去渲染。

21.3　编辑框实例分析

编辑框几乎涉及 UI 体制的所有方面。本节以编辑框为例来整体分析 Android UI 体系的工作机制。

21.3.1　ViewRoot 获得系统侧代理对象

ViewRoot 是 Android View 体系的中枢，它不仅要与到系统侧 WindowManageSevice、

InputMethodManagerService 等系统组件交互而且要及时触发 View Tree 的相关动作，涉及 Android View 体系的方方面面。要和系统交互就需要获得系统侧组件的代理。本节以 class InputMethodManagerService 为例分析其在 ViewRoot 里的代理生成。

另外，除了分析这种 service 代理对象的生成，本节还将分析非 service 对象的生成实例。

没有本地对象就没有代理对象，首先分析系统侧本地对象的生成，系统中 SystemServer 创建系统级输入服务 class InputMethodManagerService，并将其注册到 serviceManager 中，代码如下：

```
ServerThread. run() {
          try {
             …
             imm = new InputMethodManagerService(context, statusBar);
             ServiceManager.addService(Context.INPUT_METHOD_SERVICE,
imm);
/*创建 InputMethodManagerService，并将其放入 ServiceManager。系统或其他组件通过
  向 ServiceManager 查询 Context.INPUT_METHOD_SERVICE 来获得 class InputMethod
  ManagerService 的 proxy*/
          } catch (Throwable e) {
             …
          }
}
```

在应用侧，class viewroot 中生成其代理，代码如下：

```
   public static IWindowSession getWindowSession(Looper mainLooper) {
    …
    /*InputMethodManagerService 代理生成，这是通过 servicemanager 获得代理对象
      的机制。WindowManagerService 等服务的代理也是通过这种方式进行的。而对于一些
      非服务对象的代理的获取，如 WindowSession 代理对象的生成，是通过对 Window
      ManagerService 远程调用来完成。在内核解包分析的时候将在 ServiceServer 进程底
      下生成 WindowSession 本地节点，然后将 Binder 数据包挂到应用进程的时候将
      WindowSession 对象句柄改为 BINDER_TYPE_HANDLE 或 BINDER_TYPE_WEAK_
      HANDLE 类型，且在应用进程中生成引用，并指向 ServiceServer 进程的下节点。具体分
      析请参见 Binder 章节*/
    InputMethodManager imm = InputMethodManager.getInstance(mainLooper);
    …
    }
```

应用中的 class InputMethodManager 实例构建及典型的 service 代理对象的生成如下：

```
   static public InputMethodManager getInstance(Looper mainLooper) {
      synchronized (mInstanceSync) {
         …
//先到 servicemanager 里找到 InputMethodManagerService
  IBinder b = ServiceManager.getService(Context.INPUT_METHOD_SERVICE);
//生成 InputMethodManagerService 的代理对象
```

```
            IInputMethodManager   service   =   IInputMethodManager.Stub.
asInterface(b);
            mInstance = new InputMethodManager(service, mainLooper);
        }
        return mInstance;
    }
```

21.3.2　焦点切换事件——主要 Android UI 机制的互动

焦点切换事件最能体现 Android UI 机制的工作机制，它几乎涉及所有主要的 Android UI 机制：WindowManagerService、InputMethodManagerService、ViewRoot、ViewTree，以及 ViewTree 的集成与重载关系。本节以焦点切换事件的处理为例分析这些机制的互动关系。

首先 WINDOW_FOCUS_CHANGED 消息被 WindowManagerService 发送给 class ViewRoot。在 class ViewRoot 里，WINDOW_FOCUS_CHANGED 消息处理流程如下：

```
public void handleMessage(Message msg) {
......
case WINDOW_FOCUS_CHANGED: {
              …
              InputMethodManager imm = InputMethodManager.peekInstance();
              if (mView != null) {
                  //一方面向 ViewTree 分发焦点改变事件
                  mView.dispatchWindowFocusChanged(hasWindowFocus);
              }
              …
              /*另一方面如果获得焦点，则有可能需要提供输入操作，这里将整棵 View Tree
                交给 InputMethodManager 来处理*/
              if (hasWindowFocus) {
                  if (imm != null && mLastWasImTarget) {
               …
              /*调用 InputMethodManager 的 onWindowFocus 方法来处理 FOCUS 事件。
                这里参数 mView 为 ViewTree*/
                      imm.onWindowFocus(mView, mView.findFocus(),
                              mWindowAttributes.softInputMode,
                              !mHasHadWindowFocus,       mWindowAttributes.
flags);
                  }
               ......
        } break;
```

当 WINDOW_FOCUS_CHANGED 事件被转发到 InputMethodManager 时，一系列函数被触发，代码如下：

```
public void onWindowFocus(…) {
```

```
            //一方面，向 ViewTree 分发焦点检查事件
            focusInLocked(focusedView != null ? focusedView : rootView);
            /*另一方面，对于 InputMethodManager 自身要检查焦点，一旦确认获得焦点需要启
              动输入法*/
            checkFocus();

        }
    //焦点检查函数
    public void checkFocus() {
           …
            /*启动输入法，InputMethodManager 自身有一个系统侧 class InputMethod
              ManagerService 的代理对象，启动相关输入法工作将通知该组件去完成*/
           startInputInner();
        }
```

而在 View 体系中，当一个需要使用输入法获得输入的 View 在收到焦点改变的消息时，都要调用其父类 onFocusChanged 函数，代码如下：

```
    Class View::onFocusChanged(…)
    {…
            imm.focusIn(this);
    …
    }
    //这里告诉输入法要把输入送到哪个 View
    Class InputMethodManager:: focusInLocked(View view)
    {…
             mNextServedView = view;
     …
    }
```

21.3.3 输入事件的处理

在处理完焦点切换事件之后，整个 UI 机制就可以进行输入操作了。当触摸事件或鼠标事件首先被 windowmanagerservice 从内核 input 驱动读进来之后，系统侧的 windowmanagerservice 找到当前最顶层窗口对应的 ViewRoot，将事件分发到应用侧。

```
    //触摸事件被送进 viewroot
    private void deliverKeyEvent(KeyEvent event, boolean sendDone) {
            …
          //首先让自己的 ViewTree 来处理该触摸事件
            boolean handled = mView != null
                    ? mView.dispatchKeyEventPreIme(event) : true;
            if (handled) {
                …
                //如果该事件被 ViewTree 中的某个 View 处理了，则直接返回
                return;
```

```
        }
        …
        /*否则就把触摸事件交给 InputMethodManager，接下来 InputMethodManager 通
          过代理对象 IInputMethodManager mService;把事件提交给系统侧的输入法管理
          器，再由其与具体的输入法交互*/
        if (mLastWasImTarget) {
                  …
                  imm.dispatchKeyEvent(mView.getContext(), seq, event,
                          mInputMethodCallback);
                  return;
            }
        }
        …
}
```

21.3.4　编辑框的生成

本节以 MMS 应用为例分析编辑框生成。在使用 MMS 应用编辑一个短消息时会启动一个 activity，这个 activity 位于 packages/apps/Mms/src/com/android/mms/ui/ComposeMessageActivity.java，描述其 UI 的 xml 文件为 compose_message_activity.xml，里面值得研究的是短信接收人编辑框。

在 packages/apps/Mms/res/layout/compose_message_activity.xml 按如下方式定义了短信接收人编辑框：

```
<ViewStub android:id="@+id/recipients_editor_stub"
    android:layout="@layout/recipients_editor"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
/>
```

可见，compose_message_activity.xml 并没有采用普通指定类名，而是采用 ViewStub 来描述编辑框。其中 layout 指向 packages/apps/Mms/res/layout/recipients_editor.xml。

```
<com.android.mms.ui.RecipientsEditor
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/recipients_editor"
    …
    />
```

com.android.mms.ui.RecipientsEditor 指出了类信息。接下来看这个编辑框是如何生成。

```
//首先看初始化函数
  private void initialize(Bundle savedInstanceState) {


         if (mConversation.getThreadId() <= 0) {
                    //仅分析接收人为空的情况
                    initRecipientsEditor();
```

```
        } else {
            …
        }

    //编辑框初始化
    private void initRecipientsEditor() {
      ViewStub stub = (ViewStub)findViewById(R.id.recipients_editor_stub);

        if (stub != null) {
            //将编辑框 recipients_editor 给 inflate 出来
            mRecipientsEditor = (RecipientsEditor) stub.inflate();
        } else {
            …
        }
    }
```

class ViewStub 本质上就是个 class View，一个 View 使用 class ViewStub 代替自己在 UI 框架中占个位置，自己然后再动态地替换掉 ViewStub。

```
    public View inflate() {
            //找到 ViewStub 的父 View
            final ViewParent viewParent = getParent();

            if (viewParent != null && viewParent instanceof ViewGroup) {
                if (mLayoutResource != 0) {
                    final ViewGroup parent = (ViewGroup) viewParent;
                  /*找到系统中 LayoutInflater，其工作就是 inflate，跟普通的 View
    inflate 一样，只不过这里指定父 View，不用 Inflater 再为自己包层壳了*/
                  final LayoutInflater factory = LayoutInflater.from(mContext);
                  /*执行 inflate，就是根据指定的资源生成相应的 View*/
                  final View view = factory.inflate(mLayoutResource, parent,
                          false);

                  //把 ViewStub 从父 View 里拿掉
                  final int index = parent.indexOfChild(this);
                  parent.removeViewInLayout(this);

                  //把新生成 View 加到父 View 中
                  final ViewGroup.LayoutParams layoutParams = getLayout
                  Params();
                  if (layoutParams != null) {
                       parent.addView(view, index, layoutParams);
                  } else {
                       parent.addView(view, index);
                  }
                  ……
        }
```

第 22 章　ADB

ADB 尽管只是 Android 系统的一个调试工具，但是 ABD 加以改造，可以在其基础上做出非常实用的系统功能。本文仅分析 ADB 手机端的行为，且仅分析通过网络连接情景，不过 ADB 在 PC 端和手机端使用一套代码，PC 侧的行为可参照手机侧分析。

22.1　ADB 基本结构

22.1.1　连接

本节分析手机侧 ADB 与远端建立连接的过程。

```
int adb_main(int is_daemon, int server_port)
{
  …
  /* 创 建  unix_socket  句 柄 对  transport_registration_send 、 transport_registration_recv，用户来管理连接*/
    init_transport_registration();
    …

    //检查 init.rc 中的"service.adb.tcp.port"是否置位
    property_get("service.adb.tcp.port", value, "");

    if (!value[0])
        property_get("persist.adb.tcp.port", value, "");
    if (sscanf(value, "%d", &port) == 1 && port > 0) {
        /*若"service.adb.tcp.port"或"persist.adb.tcp.port"指定了 tcp port，那
          么 ADB 通过网络连接。void local_init(int port)fork 出连接线程，该线程是
          典型的 linux server 线程：static void *server_socket_thread(void * arg)*/
        local_init(port);
    } else if (access("/dev/android_adb", F_OK) == 0) {
        //否则如果存在/dev/android_adb 设备，则通过 USB 连接
        usb_init();
    } else {
        //否则通过默认网络端口连接
        local_init(DEFAULT_ADB_LOCAL_TRANSPORT_PORT);
}
```

```
    …
    //主线程循环
    fdevent_loop();
    return 0;
}

//连接线程
static void *server_socket_thread(void * arg)
{
    int serverfd, fd;
    struct sockaddr addr;
    socklen_t alen;
    int port = (int)arg;
     …
    serverfd = -1;
    for(;;) {
        if(serverfd == -1) {
            //socket 调用
            serverfd = socket_inaddr_any_server(port, SOCK_STREAM);
              …
            close_on_exec(serverfd);
        }

        alen = sizeof(addr);
        //在该端口上等待对方的 connect
        fd = adb_socket_accept(serverfd, &addr, &alen);
        …
         //成功连接，取得句柄
        if(fd >= 0) {
            …
         /*创建代表该连接的 struct atransport 结构，该结构最关键的成员变量为：int sfd;
           ——记录了该 socket 句柄
int (*read_from_remote)(...);——int remote_read(...)通过该 socket 从远端读
int (*write_to_remote)(...);——int remote_write(...)通过该 socket 向远端写
然后该函数通过 transport_registration_send 向连接管理机构注册该 struct
atransport 结构*/
            register_socket_transport(fd, "host", port, 1);
        }
    }
    …
}
```

连接管理函数 static void transport_registration_func(…)运行在主线程中，对于每一个 struct atransport 结构都创建一个 input_thread 和一个 output_thread，见下文分析。

22.1.2　主线程

ADB 的主线程是 void fdevent_loop()，该线程通过 select 机制不断检测自己关注的若干文件句柄上是否有事件发生，若有事件发生，将为对应文件句柄执行处理函数。代码如下：

```
void fdevent_loop()
{  …
   for(;;) {
        /*检测文件句柄上是否有事件发生，将有事件发生的句柄收集在 list_pending 链表中*/

        fdevent_process();
       //依次取出 list_pending 链表中的句柄，执行其处理函数
        while((fde = fdevent_plist_dequeue())) {
             fdevent_call_fdfunc(fde);
        }
    }
}

static void fdevent_process()
{
     //收集文件句柄位图集
     memcpy(&rfd, &read_fds, sizeof(fd_set));
     memcpy(&wfd, &write_fds, sizeof(fd_set));
     memcpy(&efd, &error_fds, sizeof(fd_set));

     //Linux 系统 select 调用
     n = select(select_n, &rfd, &wfd, &efd, NULL);

   //有事件发生
    for(i = 0; (i < select_n) && (n > 0); i++) {
          events = 0;
          //收集该句柄对应的事件
          if(FD_ISSET(i, &rfd)) { events |= FDE_READ; n--; }
          if(FD_ISSET(i, &wfd)) { events |= FDE_WRITE; n--; }
          …

        if(events) {
              fde = fd_table[i];
              …
              fde->state |= FDE_PENDING;
              //将有事件发生的句柄记录在 list_pending
              fdevent_plist_enqueue(fde);
        }
    }
```

22.1.3 主线程监测的文件句柄

1. transport_registration_recv

与 int transport_registration_send 组成 unix_socket 句柄对，触发函数为 static void transport_registration_func(int _fd, unsigned ev, void *data)，用来管理连接注册。

2. struct atransport 成员变量 transport_socket

与 struct atransport 成员变量 int fd;组成 unix_socket 句柄对，触发函数为 static void transport_socket_events(int fd, unsigned events, void *_t)，用来管理 struct atransport 的两个工作线程：static void *input_thread(void *_t)和 static void *output_thread(void *_t)。

3. 本地 struct asocket 成员变量 int fd

与本地 struct asocket 对于的服务线程组成 unix_socket 句柄对，当服务线程侧往这个 unix_socket 写操作时 static void local_socket_event_func(...)被触发，导致远端对等 struct asocket 结构的 enqueue 等函数被调用。

ADB 还包括 struct atransport 的两个工作线程：static void *input_thread(void *_t)和 static void *output_thread(void)，以及对应特定命令服务线程，这些线程的分析在后面章节介绍。

22.2 Transport

Transport 的主要结构是一个 struct atransport 和两个 struct asocket 对等对。

22.2.1 初始化

Transport 初始化的主要工作包括传输机构的注册和传输线程的创建，这些工作由框架函数 transport_registration_func 实现。

```
//Transport 框架函数
static void transport_registration_func(int _fd, unsigned ev, void *data)
{
    …
    /*这个 fd 即为 static int transport_registration_recv，与 static int
      transport_registration_send 组成一对 unix_socket，这里读取 static void
      *server_socket_thread(void * arg)传送过来的 transport 参数（对于从网络连接
      ADB 而言）*/

    if(transport_read_action(_fd, &m)) {
```

```
        fatal_errno("cannot read transport registration socket");
    }

//static void *server_socket_thread(void * arg)创建的struct atransport 结构
    t = m.transport;
    …
    if (t->connection_state != CS_NOPERM) {
        /* initial references are the two threads */
        t->ref_count = 2;
        /*对于一个新注册的 struct atransport，transport 管理机构为其创建两个线程：
          static void *input_thread(void *_t)和 static void *output_
          thread(void*_t)，并且创建一个 unix_socket 句柄对，用来与这两个线程通信*/
        if(adb_socketpair(s)) {
            fatal_errno("cannot open transport socketpair");
        }
        // transport_socket 是 transport 管理机构侧使用的句柄
        t->transport_socket = s[0];
        // t->fd 是另外两个新线程使用的句柄
        t->fd = s[1];

       /*若另外两个新线程往 unix_socket 写入内容，则 static void transport_
         socket_events(…)将被触发*/

        fdevent_install(&(t->transport_fde),
                        t->transport_socket,
                        transport_socket_events,
                        t);

        //将 select 检测的文件集相应位置位
        fdevent_set(&(t->transport_fde), FDE_READ);

        //创建 input_thread 线程
        if(adb_thread_create(&input_thread_ptr, input_thread, t)){
            fatal_errno("cannot create input thread");
        }

        //创建 output_thread 线程
        if(adb_thread_create(&output_thread_ptr, output_thread, t)){
            fatal_errno("cannot create output thread");
        }
    }

    /* 将该 atransport 串起来 */
    adb_mutex_lock(&transport_lock);
```

```
        t->next = &transport_list;
        t->prev = transport_list.prev;
        t->next->prev = t;
        t->prev->next = t;
       …
    }
```

22.2.2 transport 传输线程

每个 struct atransport 有 input 和 output 两个线程，这是 struct atransport 与对方通信的主要工作线程。

```
/*从远端到 transport 管理函数 static void transport_socket_events(…)*/

static void *output_thread(void *_t)
{
     atransport *t = _t;
     apacket *p;
     //分配一个 struct apacket
     p = get_apacket();
     p->msg.command = A_SYNC;
     …
    //向 static void transport_socket_events(…)发送一个 A_SYNC 命令
    if(write_packet(t->fd, t->serial, &p)) {
        …
    }

    //读取远端数据包，再传给 transport
    for(;;) {
        p = get_apacket();
        /*对于网络连接的 ADB，该函数为 static int remote_read(…)，其使用 accept
          来的与远端 socket 链接的句柄 int sfd;，从远端读取数据包*/
        if(t->read_from_remote(p, t) == 0){
            …
            //将取来的 packet 写到 static void transport_socket_events(…)
            if(write_packet(t->fd, t->serial, &p)){
               …
            }
        } else {…
        }
    }

    //作为协议的一部分再写一个 A_SYNC 命令
    p = get_apacket();
    p->msg.command = A_SYNC;
```

```
    …
    if(write_packet(t->fd, t->serial, &p)) {
        …
    }
…
}

//将数据流从 transport 管理函数 static void transport_socket_events(…)传输到远端
static void *input_thread(void *_t)
{
     atransport *t = _t;
     apacket *p;
     int active = 0;

    …
    for(;;){
    //从 static void transport_socket_events(…)读取数据包
        if(read_packet(t->fd, t->serial, &p)) {
            …
        }



        if(p->msg.command == A_SYNC){
             //如果是 A_SYNC 则调整自己状态
            …
        } else {
            if(active) {
                /*将数据写向远端，对于网络连接的 ADB，即调用函数 static int remote_
                  write(…)，该函数使用 accept 来的远端 socket 链接的句柄 int sfd;，
                 向远端写数据包*/
                t->write_to_remote(p, t);
            } else {
               …
            }
        }
…
    }

    …
    return 0;
}
```

22.2.3 transport 的管理

当远端的数据到来，select 的 transport_fde 句柄上有事件发生，导致 static void transport_socket_events(…)被触发。

```
static void transport_socket_events(int fd, unsigned events, void *_t)
{
    …
    if(events & FDE_READ){
        apacket *p = 0;
        //接收自己 input_thread，output_thread 传送来的数据包
        if(read_packet(fd, t->serial, &p)){
            …
       } else {
       //处理接收到的数据包
            handle_packet(p, (atransport *) _t);
      }
   }
}

/*数据包的处理，实现与远端的交互协议，ADB 交互协议参见相关文档，这里仅以从 PC 通过以太
  网连接 target 并 push 一个文件到板子的情景来分析*/
  void handle_packet(apacket *p, atransport *t)
{
   asocket *s;
   …
   switch(p->msg.command){
case A_SYNC:
    /* 第一步，void handle_packet(…)与自己的 input、output 线程的同步，通过 A_SYNC
       命令来实现*/
    …
   case A_CNXN: /* CONNECT(version, maxdata, "system-id-string") */
       /* 第二步，收到 A_CNXN 包携带 connect 命令而来，这一步的处理比较简单，主要向对
          方也发送一个 A_CNXN 包即可*/
       if(t->connection_state != CS_OFFLINE) {
            t->connection_state = CS_OFFLINE;
            handle_offline(t);
       }
       parse_banner((char*) p->data, t);
       handle_online();
       if(!HOST) send_connect(t);
       break;

case A_OPEN: /* OPEN(local-id, 0, "destination") */
```

```
    /* 第三步，push 命令导致一个 A_OPEN 包发送到 target*/

        if(t->connection_state != CS_OFFLINE) {
           //push 命令的数据包里指定服务名为 sync:
            char *name = (char*) p->data;
           //整理服务名
            name[p->msg.data_length > 0 ? p->msg.data_length - 1 : 0] = 0;
            //asocket *create_local_service_socket(…)分析见下文

            s = create_local_service_socket(name);
            if(s == 0) {
                send_close(0, p->msg.arg0, t);
            } else {
               /*为本地 struct asocket 创建一个对等结构，代表远端，远端送来的数据包指
                 定自己的 ID，p->msg.arg0*/
                s->peer = create_remote_socket(p->msg.arg0, t);
                s->peer->peer = s;
                //ACK 远端一个 A_OKAY
                send_ready(s->id, s->peer->id, t);
                s->ready(s);
            }
        }
        break;

case A_OKAY: /* READY(local-id, remote-id, "") */
    //第四步，远端发来 A_OKAY
        if(t->connection_state != CS_OFFLINE) {
            //检查双方的对等 struct apacket 是否建立
            if((s = find_local_socket(p->msg.arg1))) {
                if(s->peer == 0) {
                        s->peer = create_remote_socket(p->msg.arg0, t);
                        s->peer->peer = s;
                }
                s->ready(s);
            }
        }
        break;
…
case A_WRTE:
//第五步，反复重复的步骤，数据传输主要由 A_WRTE 数据包携带
        if(t->connection_state != CS_OFFLINE) {
              //寻找 struct asocket 对的本地部分
              if((s = find_local_socket(p->msg.arg1))) {
                     unsigned rid = p->msg.arg0;
                     p->len = p->msg.data_length;
```

```
                    //执行其 int (*enqueue)(...);函数，分析详见下文
                    if(s->enqueue(s, p) == 0) {
                        …
                        send_ready(s->id, rid, t);
                    }
                    return;
                }
            }
          …
        }
…
}


asocket *create_local_service_socket(const char *name)
{
    asocket *s;
    int fd;
    ...
    /*通过 int service_to_fd(const char *name)创建 sync:的服务线程，并通过一个
unix_socket 与之通信，返回句柄 fd，即为该 unix_socket 的句柄*/
    fd = service_to_fd(name);
    if(fd < 0) return 0;

   /*创建一个 struct asocket 结构，并将其加入 local_socket_list 队列，该 struct
     asocket 结构关键成员函数 int fd 即为与上述 sync:的服务线程通信的 unix_socket 的
    句柄，其成员函数 int (*enqueue)(…);为 static int local_socket_enqueue(…)*/
   s = create_local_socket(fd);
   //返回该 struct asocket 结构指针
    return s;
}
//建立本地 asocket
asocket *create_local_socket(int fd)
{
    asocket *s = calloc(1, sizeof(asocket));
    //控制服务线程的 unix_socket
    s->fd = fd;
    //数据包来到时调用
    s->enqueue = local_socket_enqueue;
    s->ready = local_socket_ready;
    s->close = local_socket_close;
    //加入本地 local_socket_list 链表
    install_local_socket(s);
    /*当服务线程要向远端写入数据时，static void local_socket_event_func(…)得到
      调用，该函数将调用本地 asocket 的远端对等 asocket 的 int (*enqueue)(…);函数，
```

```
    使得服务线程传来的数据送到远端*/
    fdevent_install(&s->fde, fd, local_socket_event_func, s);
    return s;
}


/*在每次 A_WRTE 到来时,对应本地的 struct asocket 结构的 int (*enqueue)(asocket *s,
  apacket *pkt);被触发*/
static int local_socket_enqueue(asocket *s, apacket *p)
{
  …
    //若该包携带数据
    while(p->len > 0) {
    //向服务线程写入数据，s->fd 对应服务线程段的 unix_socket 句柄对
        int r = adb_write(s->fd, p->ptr, p->len);
        …
    }

    …

}
```

22.3　Local 服务

22.3.1　Local 服务的种类

Local 指的是手机或嵌入式系统上的特定功能。

```
//Local  service 通过字符串 name 来索引
int service_to_fd(const char *name)
{
     int ret = -1;
     …

#if ADB_HOST
  …
#else /* !ADB_HOST */
    } else if(!strncmp("dev:", name, 4)) {
        ret = unix_open(name + 4, O_RDWR);
} else if(!strncmp(name, "framebuffer:", 12)) {
        //framebuffer 服务，抓屏
        ret = create_service_thread(framebuffer_service, 0);
```

```
} else if(recovery_mode && !strncmp(name, "recover:", 8)) {
        //recover 服务，升级
        ret = create_service_thread(recover_service, (void*) atoi(name +
        8));
} else if (!strncmp(name, "jdwp:", 5)) {
        //jdwp 链接 PC 调试器
        ret = create_jdwp_connection_fd(atoi(name+5));
} else if (!strncmp(name, "log:", 4)) {
    //抓 log
        ret=create_service_thread(log_service, get_log_file_path(name +
        4));
} else if(!HOST && !strncmp(name, "shell:", 6)) {
    //连 shell
        if(name[6]) {
            ret = create_subproc_thread(name + 6);
        } else {
            ret = create_subproc_thread(0);
        }
} else if(!strncmp(name, "sync:", 5)) {
     //最常用的文件传输服务，后续章节详细分析
        ret = create_service_thread(file_sync_service, NULL);
} else if(!strncmp(name, "remount:", 8)) {
  //重新 mount/system 目录
        ret = create_service_thread(remount_service, NULL);
} else if(!strncmp(name, "reboot:", 7)) {
  //重启，要等 vdc 完成相关工作之后才能执行重启
        void* arg = strdup(name + 7);
        if(arg == 0) return -1;
        ret = create_service_thread(reboot_service, arg);
} else if(!strncmp(name, "root:", 5)) {
    /*通过检查 ro.debuggable 来决定是否置位 service.adb.root，以使得 adb root。手
      机系统上并不是每次都能成功，要看出厂设置*/
        ret = create_service_thread(restart_root_service, NULL);
} else if(!strncmp(name, "backup:", 7)) {
//备份，通过/system/bin/bu 来完成
        char* arg = strdup(name+7);
        if (arg == NULL) return -1;
        ret = backup_service(BACKUP, arg);
} else if(!strncmp(name, "restore:", 8)) {
//恢复，也是通过/system/bin/bu 来完成
        ret = backup_service(RESTORE, NULL);
} else if(!strncmp(name, "tcpip:", 6)) {
//通过设置 service.adb.tcp.port 触发相关动作
        int port;
        if (sscanf(name + 6, "%d", &port) == 0) {
```

```
            port = 0;
        }
        ret = create_service_thread(restart_tcp_service, (void *)port);
} else if(!strncmp(name, "usb:", 4)) {
//通过设置 service.adb.tcp.port 触发相关动作
        ret = create_service_thread(restart_usb_service, NULL);
#endif
    …
    }
    …
    return ret;
}
```

若希望通过 ADB 实现其他特殊功能，也需要在 int service_to_fd(const char *name)增加相关分支。

22.3.2　Local 服务的形态

ADB Local service 的形式是以独立线程形式存在的，然后通过 unix_socket 与之通信。

```
//创建一个 ADB service
static int create_service_thread(void (*func)(int, void *), void *cookie)
{
    stinfo *sti;
    adb_thread_t t;
    //unix_socket 的一对文件句柄
    int s[2];
    //创建 unix_socket，句柄放在 int s[2]里
    if(adb_socketpair(s)) {
        …
    }
    //stinfo *sti 用来存放参数
    sti = malloc(sizeof(stinfo));
    if(sti == 0) fatal("cannot allocate stinfo");
    sti->func = func;
    sti->cookie = cookie;
    sti->fd = s[1];

    //创建独立的服务线程
    if(adb_thread_create( &t, service_bootstrap_func, sti)){
     …
    }
    …
    return s[0];
}
```

22.3.3 SYNC 服务

以 SYNC 服务为例分析，一个 Local 服务的实现。当 Push 一个文件到 target，SYNC 服务将会被启动。

```
void file_sync_service(int fd, void *cookie)
{

    for(;;) {
        //从 UNIX_SOCKET 里读取文件名
        if(readx(fd, &msg.req, sizeof(msg.req))) {
            …
        }
        …
        if(readx(fd, name, namelen)) {
        …
        }
        //name 中即为 target 的文件名
        name[namelen] = 0;
        …
        // Push 一个文件的写入操作，对应的 ID 为 ID_SEND
        switch(msg.req.id) {
        case ID_STAT:
            …
        case ID_SEND:
            /*在处理完链接操作之后，文件接收的主题通过 static int handle_send_
              file(…)来实现，这里将 UNIX_SOCKET 句柄传送过去*/
            if(do_send(fd, name, buffer)) goto fail;
            break;
        case ID_RECV:
            …
        }
    }

//UNIX_SOCKET 句柄 s 是连接远端的纽带
static int handle_send_file(int s, char *path, mode_t mode, char *buffer)
{
    …
   //文件打开操作
    fd = adb_open_mode(path, O_WRONLY | O_CREAT | O_EXCL, mode);
    if(fd < 0 && errno == ENOENT) {
        mkdirs(path);
        fd = adb_open_mode(path, O_WRONLY | O_CREAT | O_EXCL, mode);
    }
    …
```

```
    //文件接受
    for(;;) {
            …
            //读取包头
           if(readx(s, &msg.data, sizeof(msg.data)))
                 goto fail;
           //检查是否为数据包
           if(msg.data.id != ID_DATA) {
               //是否到了文件结束
               if(msg.data.id == ID_DONE) {
                 timestamp = ltohl(msg.data.size);
                 break;
             }
            …
        }
       …
       /*读取数据,数据流经路径为:从 static void *output_thread(void *_t)到 struct
         asocket 的 int (*enqueue)(...);函数 (static int local_socket_
         enqueue(…)), 再到这里*/
        if(readx(s, buffer, len))
             goto fail;
        …
        //写入 target 本地文件系统
        if(writex(fd, buffer, len)) {
            ...
        }
    }
    //文件写入成, ACK 远端
 if(fd >= 0) {
      //ACK 包将携带文件戳等信息到远端
        struct utimbuf u;
        adb_close(fd);
        u.actime = timestamp;
        u.modtime = timestamp;
        utime(path, &u);
        //状态置为 OK
        msg.status.id = ID_OKAY;
        msg.status.msglen = 0;
        //写到远端
        if(writex(s, &msg.status, sizeof(msg.status)))
            return -1;
    }
     return 0;
 …
 } }
```

第 23 章　Android 浏览器的 Webkit 分析

随着 HTML5 的流行，提供一个好的 Webruntime 支持成为 OS 的必选项。Webkit 由两部分组成：Webcore 和 js 引擎。前者的工作是渲染 html 脚本，后者是 js 执行机构。有多种 Js 引擎实现，本书着重于 V8 版本。

本章 Webcore 分析的代码版本为 Android 2.x 集成的 Webkit，V8 部分由于在 PC 上调试较容易，取自独立的 V8 代码库，版本为 2.2.14。

23.1　Webcore

Webcore 侧的工作由 html 语法解析、Dom 树的构建、rendering tree 的构建、layout、事件接收组成。其基本工作机理为：Webcore 将 html 节点做语法分析之后构建以 html 语法节点为节点的 DOM 树，然后结合 CSS 生成其 rendering tree，该 rendering tree 上的每个节点与 DOM 树相对应，是其渲染的体现形式。然后 Webcore 从 rendering tree 根节点开始逐个画出每个 rendering 节点，其中每个 rendering 节点都有自己的渲染入口。事件分发亦是以 DOM 树的逻辑结构和 rendering tree 位置信息为依据，将事件分发到对应节点上。

23.1.1　DOM 与 Rendering 树生成

在经历 html 语法解析之后，html 脚本中的语言元素得以 token 的形式表示，HTMLParser::parseToken(…)函数将逐一处理这些 token，针对不同类型的 token 生成其 DOM 节点，并逐个挂入 DOM 树上，同时根据 DOM 节点生成 Rendering 节点。这样 DOM 树和 Rendering 树同时生长，直到 token 被处理完。

```
//语法元素作为输入，生成 DOM 树和 Rendering 树
PassRefPtr<Node> HTMLParser::parseToken(Token* t)
{
    …
    /*t 是一个元素，比如是个：<img src="home_series.png">，这里会创建 class
HTMLImageElement*/
    RefPtr<Node> n = getNode(t);
    …
    if (n->isHTMLElement()) {
        HTMLElement* e = static_cast<HTMLElement*>(n.get());
```

```
        //在 class HTMLImageElement 类型的节点里添加 src 属性
        e->setAttributeMap(t->attrs.get());
      …
    }
    /*该函数把生成节点插入 DOM 树，在 insertnode 函数中会针对每个节点调用其 attach()
      函数，从而导致该节点对应的 rendering 对象的生成*/
    if (!insertNode(n.get(), t->selfClosingTag)) {
    …
    }
    return n;
}
```

在 DOM 树构建完毕后，Webcore 根据 DOM 树生成 rendering 树，接着从这个 rendering tree 的根发起 Layout。

```
void FrameView::layout(bool allowSubtree)
{
   //发起 layout，从 rendering tree 之上而下，layout
    root->layout();
}
```

接下来以如下脚本为例，绘制其 DOM 树和 Rendering 树

```
<html>
<head>
<title>Titlep</title>
</head>
<body>
shomepage <label>Thislabel </label>wang
<label>senixlabel </label>sen
</body>
</html>
```

生成的 DOM 树如图 23-1 所示

与之对应的 rendering 生成树呈现结构如图 23-2 所示。

在计算节点位置时需要对依据 CSS 脚本的信息，CSS 的解析主要是通过 lex 和 yacc 工具完成，中间生成文件位于 android\out\target\product\generic\obj\SHARED_LIBRARIES\libwebcore_intermediates\WebCore\CSSGrammar.cpp。

```
//其中 CSS 解析的入口函数如下，主要动作分为两步
bool CSSStyleSheet::parseString(const String &string, bool strict)
{   …
    //动态生成 CSSParser 对象
    CSSParser p(strict);
    //解析获取的 CSS 字符
    p.parseSheet(this, string);
    …
}
```

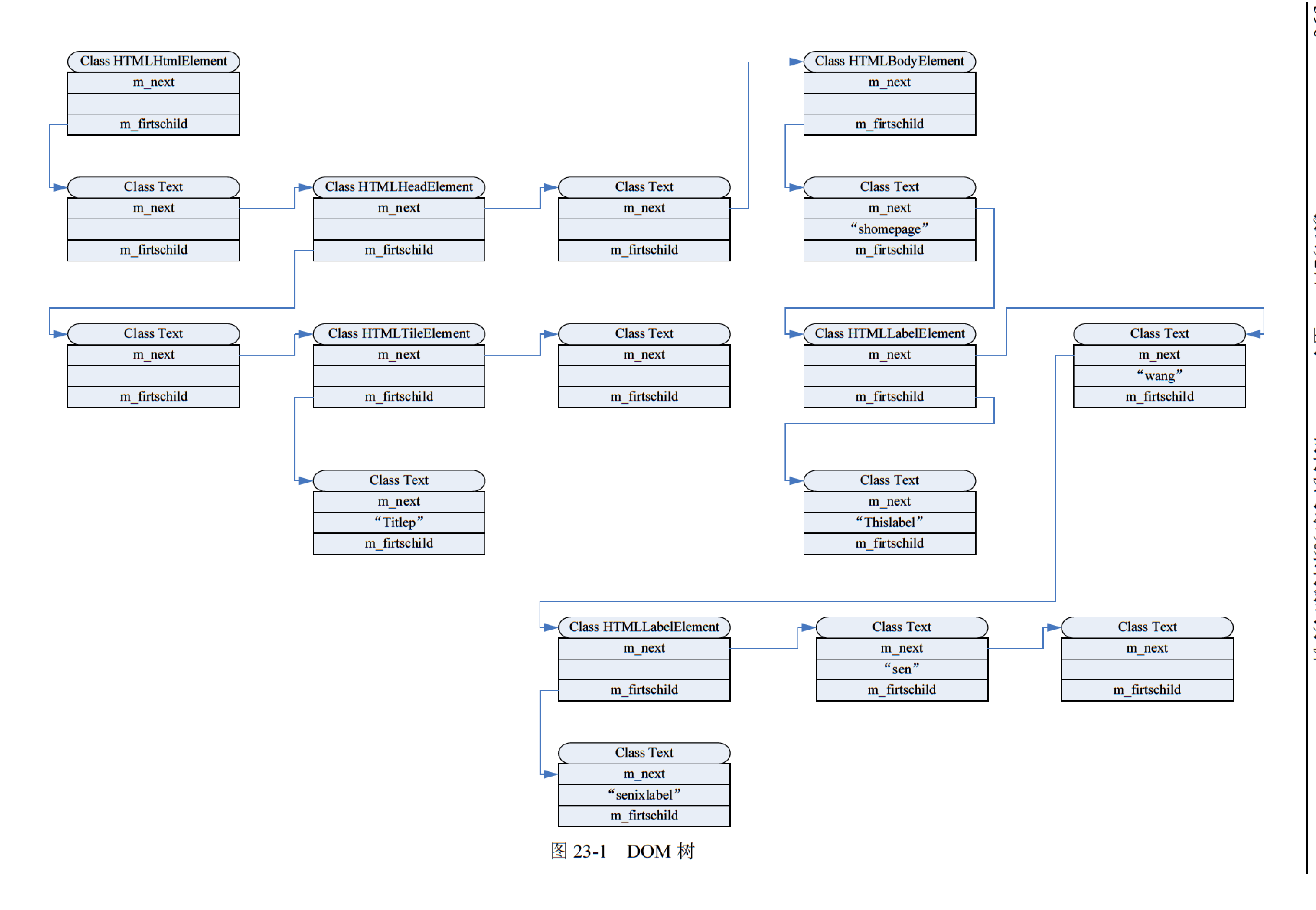

Class HTMLHtmlElement
m_next
m_firtschild
Class HTMLBodyElement
m_next
m_firtschild
Class Text
m_next
m_firtschild
Class HTMLHeadElement
m_next
m_firtschild
Class Text
m_next
m_firtschild
Class Text
m_next
“shomepage”
m_firtschild
Class Text
m_next
m_firtschild
Class HTMLTileElement
m_next
m_firtschild
Class Text
m_next
m_firtschild
Class HTMLLabelElement
m_next
m_firtschild
Class Text
m_next
“wang”
m_firtschild
Class Text
m_next
“Titlep”
m_firtschild
Class Text
m_next
“Thislabel”
m_firtschild
Class HTMLLabelElement
m_next
m_firtschild
Class Text
m_next
“sen”
m_firtschild
Class Text
m_next
m_firtschild
Class Text
m_next
“senixlabel”
m_firtschild

图 23-1　DOM 树

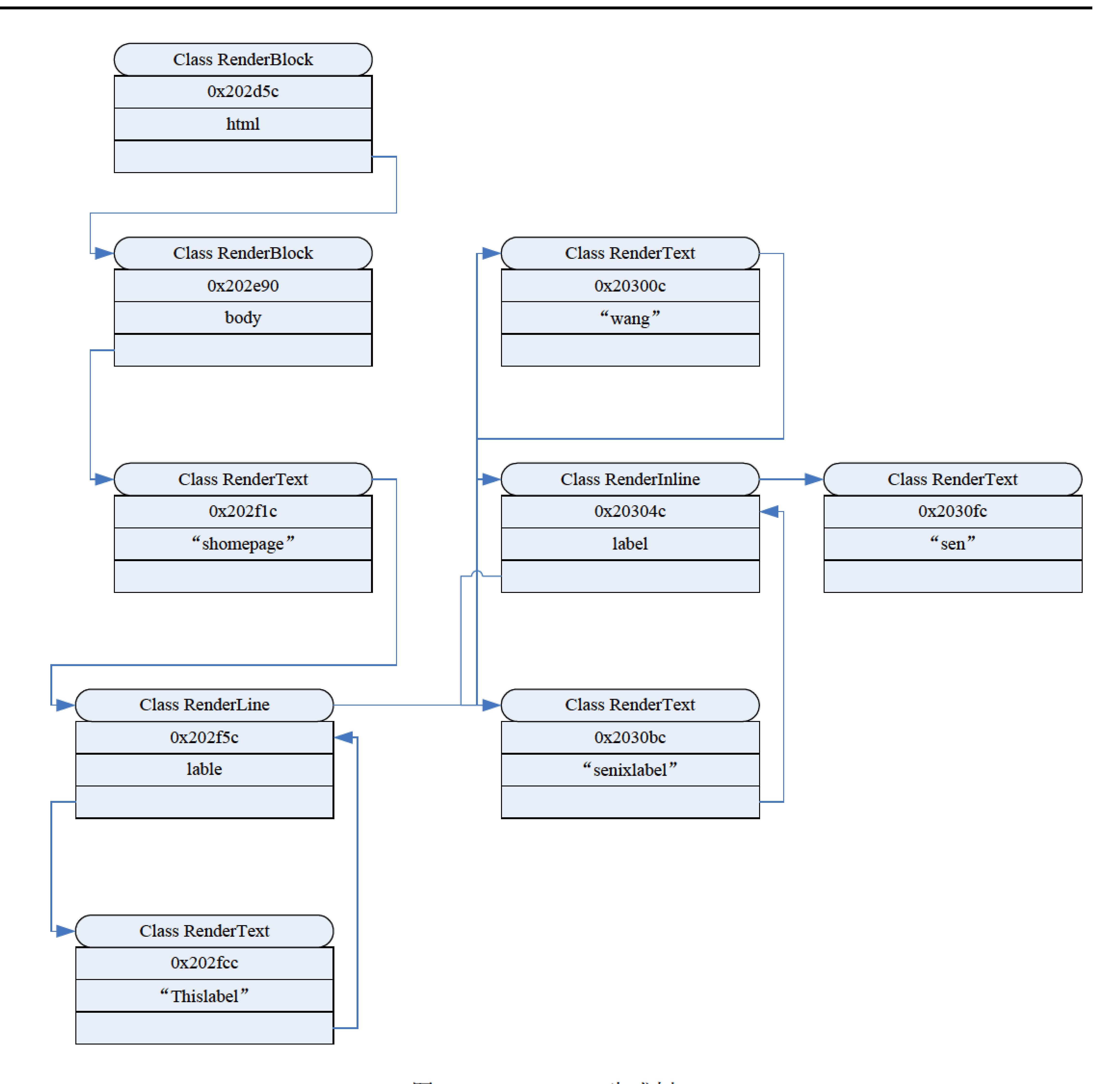


图 23-2　rendering 生成树

在 CSSParser::parseSheet 中使用 lex and yacc 工具分析 CSS 语法。

23.1.2　事件的产生与分发

首先分析事件处理函数的挂载，在构建 DOM 树时，如果该节点上有相应的处理函数，则在该节点生成以后，要在节点上挂接该节点的事件 listener。以后当 Webcore 引擎分发事件到该节点时，就会顺着注册好的 listener 找到处理函数。

事件类型是繁杂的，但是其处理流程是相同的，以一种事件处理为例可以分析出整个事件处理机制，下面以如下 script 脚本为例：

```
<html>
<head>
  <script language="LiveScript">
```

```
    function pushbutton() {
      alert("hello");
  }
 </script>

</head>
<body>
<form>
  <input type="button" name="Button1" value="Push me" onclick=
  "pushbutton()">
 </form>
</body>
</html>
```

节点处理机制的挂载要从节点生成看起，再次分析节点处理函数，这里要关注对节点的属性处理。

```
PassRefPtr<Node> HTMLParser::parseToken(Token* t)
{    …
    if (n->isHTMLElement()) {
            HTMLElement* e = static_cast<HTMLElement*>(n.get());
            //从这里进去，挂接 event 的处理 listener
            e->setAttributeMap(t->attrs.get());
            …
            }
}
```

对于每个继承自 class Element 的节点，其属性被集中存放在类型为 class NamedAttrMap 的 namedAttrMap 变量中，这里不去考虑其实现细节，将其当作一个容器即可。在构建完一个节点之后，Webcore 把从 parser 分析出来的属性列表储存到这个容器中。

```
//属性收集
void Element::setAttributeMap(PassRefPtr<NamedAttrMap> list)
{   …
    if (namedAttrMap) {
         …
        for (unsigned i = 0; i < len; i++)
        //属性更新事件
        attributeChanged(namedAttrMap->m_attributes[i].get());
    }
}
```

对于本例，parser 分析出 class HTMLInputElement 类型的节点拥有如下属性：type、name、value、onclick。在将这些属性存入 namedAttrMap 之后，还要针对每个属性触发更新函数。

```
void StyledElement::attributeChanged(Attribute* attr, bool preserveDecls)
{…
    if (needToParse)//对于某些类型的属性需要进一步处理
        parseMappedAttribute(mappedAttr);
…
}
```

继承至 class Element 有多种节点，每种节点都有自己感兴趣的属性更新。对于 class HTMLInputElement，它重载了 parseMappedAttribute 函数，显然它关心 width、height、onfocus、onselect 等属性，对于 onclick，class HTMLInputElement 并不关心，将其甩给它的父类处理。

```
void HTMLInputElement::parseMappedAttribute(MappedAttribute *attr)
{    …
     //交给父类处理
     HTMLFormControlElementWithState::parseMappedAttribute(attr);
     …
}
```

class HTMLInputElement 的父类 class HTMLFormControlElement 仍然不关心这个 onclick 属性，继续甩给其父类。

```
void HTMLFormControlElement::parseMappedAttribute(MappedAttribute *attr)
{
        …
        //真正处理的地方
        HTMLElement::parseMappedAttribute(attr);
        …
}
```

这里，class HTMLElement 处理了 onclick 属性的更新，值得注意的是：class HTMLElement 处理了 generic 的 UI 事件类属性的更新，如 onmousedown、onmouseup、onkeydown、onkeyup 等。

```
void HTMLElement::parseMappedAttribute(MappedAttribute *attr)
{
     …
        //设置 listener
        setInlineEventListenerForTypeAndAttribute(eventNames().clickEvent,
attr);
     …
}
```

所有的 DOM 树上的节点都继承于 class Node，在 class Node 中有一个存放该节点所有 listener 的容器——class NodeRareData 类型的容器。所有节点的容器都被放到类型为 class NodeRareDataMap 的更大容器里，每个 Node 的实例用自己的指针去那个更大容器里去找存放自己 listener 的 class NodeRareData 类型的容器。

setInlineEventListenerForTypeAndAttribute 的主要工作是构建一个 listener，并加入到这

个节点的 listener 容器中。

```
void    EventTargetNode::setInlineEventListenerForTypeAndAttribute(const AtomicString& eventType, Attribute* attr)
{
    /*
    构建该类型事件的 listener 且调用 class ScriptController 的 createInlineEventListener 函数来创建类型为 class JSLazyEventListener 的 listener 。注意 class JSLazyEventListener 的成员函数 void JSAbstractEventListener::handleEvent，在事件分发到该节点后，它将被触发
    */
     setInlineEventListenerForType(eventType,
document()->createEventListener(attr->localName().string(), attr->value(), this));
}
//展开分析 listener 的挂载
void EventTargetNode::setInlineEventListenerForType(const AtomicString& eventType, PassRefPtr<EventListener> listener)
{
    //把以前注册到容器中的时间 listener 除去
    removeInlineEventListenerForType(eventType);
    if (listener)
    //将新的 listener 加入到容器中
    addEventListener(eventType, listener, false);
}
```

接着分析事件的产生与分发，首先事件产生来自 Android 系统侧，自驱动到 windowmanagerservice 再到 Web 进程的 viewroot，然后通过 WebViewCore 将事件传入 Webcore。

```
//webcore 承接事件的函数
WebViewCore::finalKitFocus
{
frame->eventHandler()->handleMouseMoveEvent(mouseEvent);
}

bool    EventHandler::handleMouseMoveEvent(const    PlatformMouseEvent& mouseEvent, HitTestResult* hoveredNode)
{
    //事件定位
    MouseEventWithHitTestResults mev = prepareMouseEvent(request, mouseEvent);
    //分发
    swallowEvent = dispatchMouseEvent(eventNames().mousemoveEvent, mev.targetNode(), false, 0, mouseEvent, true);

}
```

事件正式进入到 WebCore，继续事件分发，函数如下：

```
bool EventTargetNode::dispatchGenericEvent(PassRefPtr<Event> prpEvent)
{
    handleLocalEvents(event.get(), false);
}
```

在该节点的 listener 容器中需要与 event 事件匹配的 listener，找到之后即触发其 handleEvent 函数。

```
void EventTargetNode::handleLocalEvents(Event* event, bool useCapture)
{
    …
    RegisteredEventListenerVector listenersCopy = eventListeners();
    size_t size = listenersCopy.size();
    for (size_t i = 0; i < size; ++i) {
            …
            //接到 event
            r.listener()->handleEvent(event, false);
    }
}
```

函数 eventListeners 的作用其实就是去找该节点的 listener 容器。

```
const  RegisteredEventListenerVector&  EventTargetNode::eventListeners()
const
{
    if (hasRareData()) {
         if   (RegisteredEventListenerVector*   listeners   =   rareData()
->listeners())
            return *listeners;
    }
…
}
```

在上节中可以看到 class JSLazyEventListener 类型的 listener 已经被加入到容器中，在确认其匹配的事件类型后，其 handleEvent 函数被触发。

23.2　V8 parser 源码分析

笔者曾将该节内容发表在论坛中，本节是整理精简之后的版本。（V8 版本号：2.2.14）V8 对 js 的脚本处理流程如图 23-3 所示。

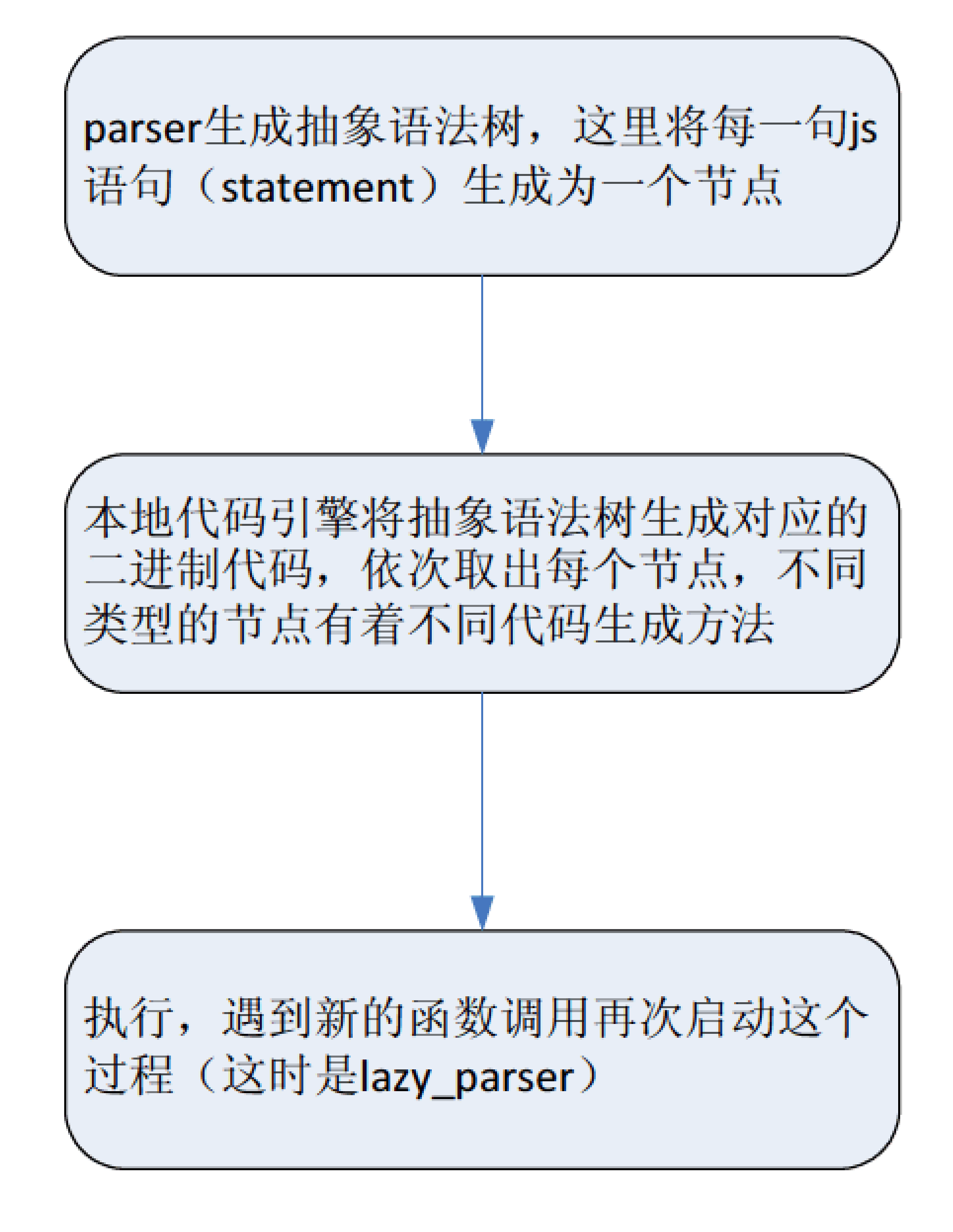


图 23-3　V8 对 js 的脚本处理流程

本节分析的正是这个流程第一步。

23.2.1　V8 parser 处理脚本的层次

（1）进入 FunctionLiteral* MakeAST(…)，这里分辨出要处理的是 json 还是普通的 js 脚本，普通的 js 脚本通过调用 parser.ParseProgram(…)处理。

（2）进入 FunctionLiteral* Parser::ParseProgram(…)，这是个 wrapper，主要工作如下：

① 构建该脚本运行时所需要的 scope，以及编译时需要的 Scanner。

② 对该脚本语法抽象树的生成工作主要由 void* Parser::ParseSourceElements(…)完成，生成的结果放在准备的容器 ZoneListWrapper<Statement> body(16)中。

③ 根据生成结果生成 FunctionLiteral 类型的返回结果。

（3）进入 void* Parser::ParseSourceElements(…)，这里的逻辑很简单，就是用 Statement* Parser::ParseStatement(…)将每行 js 语句分析出来，每条 js 语句被解析成 Statement。

```
    V8 中 Parser 定义如下：
class AstBuildingParser : public Parser {
  …
//对应脚本
Handle<Script> script_;
//脚本扫描器，将脚本输入流来处理成一个个有效元素
Scanner scanner_
//作用域，对应于变量定义作用域概念，这里代表最高层作用域
Scope* top_scope_;
…
//Expect 函数实际上将 scanner 的指针往后移动
```

```
  void Expect(Token::Value token, bool* ok);

  AstBuildingParserFactory factory_;
}

//语法分析
FunctionLiteral* Parser::ParseProgram(Handle<String> source,
                                      bool in_global_context) {
  …
// Scanner scanner_初始化
  scanner_.Initialize(source, JAVASCRIPT);
  …
  FunctionLiteral* result = NULL;
     //这里 top_scope_为 NULL
  {
    //生成一个新的 class Scope，其 parent 为 NULL
Scope* scope = factory()->NewScope(top_scope_, type, inside_with());
    /* LexicalScope 的构造函数里将 class parser 的 top_scope_指针指向刚生成 scope*/
    LexicalScope lexical_scope(this, scope);
    TemporaryScope temp_scope(this);
    ZoneListWrapper<Statement> body(16);
    bool ok = true;

    //对于最上层 top scope，其结束标志是 Token::EOS
    ParseSourceElements(&body, Token::EOS, &ok);

if (ok) {
      //生成对应于当前脚本的 FunctionLiteral
      result = NEW(FunctionLiteral(…));
    }
    …
  }
  …
}

//对于一个 Js 函数的分析
FunctionLiteral* Parser::ParseFunctionLiteral(Handle<String> var_name,
                                              int function_token_position,
                                              FunctionLiteralType type,
                                              bool* ok) {
  …
```

```
  int num_parameters = 0;
  // 函数体分析
  { Scope::Type type = Scope::FUNCTION_SCOPE;
/*进入一个函数体中，这是新一层作用域，为这个函数体新生成一个 class Scope 的实例，其父
  scope 为 top_scope_*/
    Scope* scope = factory()->NewScope(top_scope_, type, inside_with());

    /*对于本层次 class LexicalScope，在其构造函数里将其 prev_scope_指针记录着上一
      层的 top_scope_，而 class parser 的 top_scope_被指向新生成的 scope，而在 class
      LexicalScope 析构函数里将恢复 class parser 的 top_scope_为构造函数里记录的
      prev_scope_*/

    LexicalScope lexical_scope(this, scope);
    TemporaryScope temp_scope(this);
    top_scope_->SetScopeName(name);

    …
    //出现“(”，表明函数声明的开始
    Expect(Token::LPAREN, CHECK_OK);
    int start_pos = scanner_.location().beg_pos;
    bool done = (peek() == Token::RPAREN);
    //如果紧接着出现“)”，表明这个函数没有参数，否则分析其参数
    while (!done) {//分析该函数参数
      Handle<String> param_name = ParseIdentifier(CHECK_OK);
      if (!is_pre_parsing_) {
      //把参数加入到该 scope 中
      top_scope_->AddParameter(top_scope_->DeclareLocal(param_name,
      Variable::VAR));
        num_parameters++;
      }
      done = (peek() == Token::RPAREN);
      if (!done) Expect(Token::COMMA, CHECK_OK);
    }
     //终于出现“)”，该函数参数列表分析完毕
    Expect(Token::RPAREN, CHECK_OK);

    //出现“{”，一个函数体的开始
    Expect(Token::LBRACE, CHECK_OK);
/*Class ZoneListWrapper 实际上是容纳其模板类型实例的一个列表，这里其容纳的是
  Statement 列表*/
    ZoneListWrapper<Statement> body = factory()->NewList<Statement>(8);
 …
```

```
    if (!function_name.is_null() && function_name->length() > 0) {
    //生成一个变量，其名字为该函数名
      Variable* fvar = top_scope_->DeclareFunctionVar(function_name);
    //生成一个变量代理
      VariableProxy* fproxy =
          top_scope_->NewUnresolved(function_name, inside_with());
    //将变量及变量代理绑定起来
      fproxy->BindTo(fvar);
    /*首先生成一个 Assignment 表达式的实例，其 target_为 fproxy，其 value_为 class
      ThisFunction 的实例，class ThisFunction 本身就是一个表达式：class
      ThisFunction: public Expression*/

      body.Add(new ExpressionStatement(
                      new Assignment(Token::INIT_CONST, fproxy,
                                         NEW(ThisFunction()),
                                         RelocInfo::kNoPosition)));
    }
    …
    if (is_lazily_compiled && pre_data() != NULL) {
      …
  } else {
  //这里对脚本逐语句分析
      ParseSourceElements(&body, Token::RBRACE, CHECK_OK);
      …
      this_property_assignments = temp_scope.this_property_assignments();
    }

    Expect(Token::RBRACE, CHECK_OK);
    int end_pos = scanner_.location().end_pos;
    …
    //parser 的结果放在这里
    FunctionLiteral* function_literal =
       NEW(FunctionLiteral(...));
    if (!is_pre_parsing_) {
    function_literal->set_function_token_position(function_token_
    position);
    }
    seen_loop_stmt_ = true;
    return function_literal;
  }
}
```

23.2.2 Scope

在 parser 分析到一个变量定义时，以一个函数内的变量声明并赋值为例：

```
var senix;
var senix=123;
```

首先 parser 在分析到第一条语句时，语句处理函数：Block* Parser::ParseVariableDeclarations(bool accept_IN, Expression** var, bool* ok) {…}一个重要举措是在当前 scope 里 declare 这个变量。而 scope 的 declare 函数在接到这个任务之后，触发如下动作：

（1）在当前变量列表中查找是否有同名变量。

（2）如果没有同名变量，在当前 scope 中创建一个新的变量 class Variable。

（3）为该变量生成变量代理 VariableProxy，并将其加入当前 scope 的 ZoneList<VariableProxy*> unresolved_队列和 ZoneList<Declaration*> decls_队列。

（4）将该变量代理 class VariableProxy 绑定到 class Variable。

在该 scope 将生成 parser_scope_1 的结构，如图 23-4 所示。

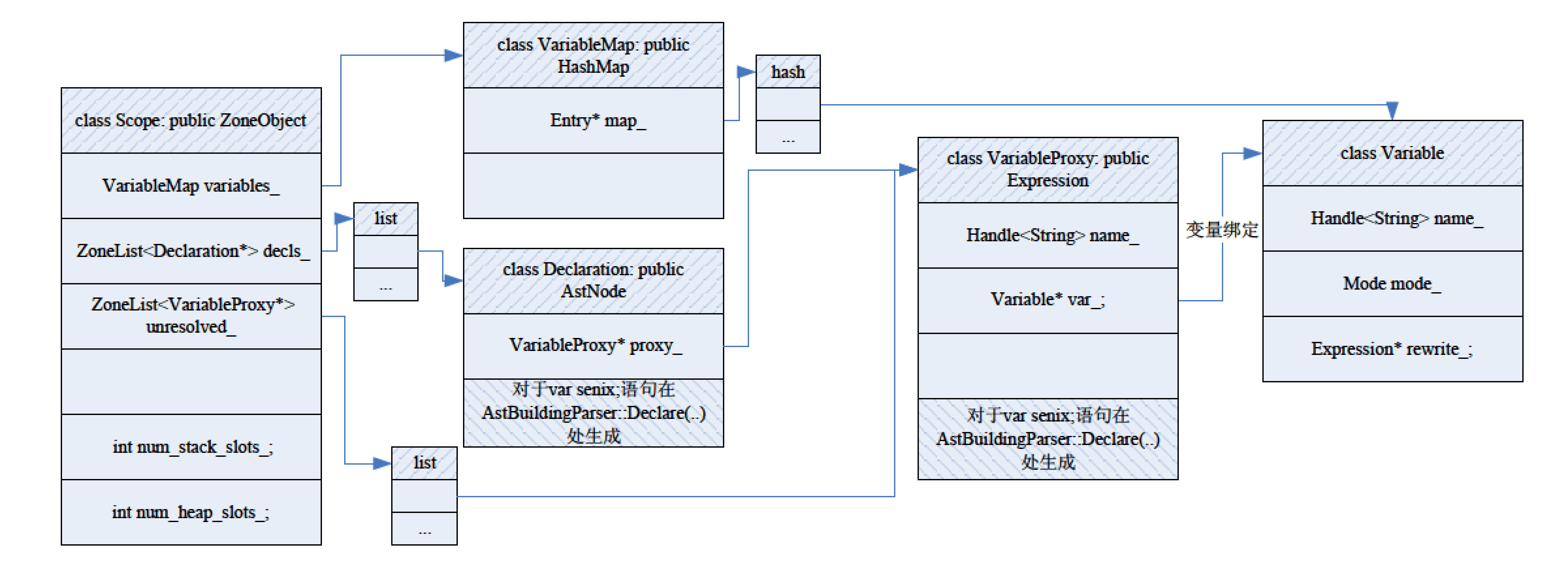


图 23-4 parser_scope_1 的结构

接着 parser 在分析到第二条语句时，这是个赋值语句，生成的赋值节点包括代表 senix 的 class VariableProxy 类型变量和代表 123 的 class Literal 变量，scope 演变成图 parser_scope_2 的结构如图 23-5 所示。

这时可以看到，第二句赋值语句仅仅指向了一个变量代理，而没有真正指向该变量。该变量代理与变量的绑定在第三步完成。

在完成了脚本 parser 以后，下一步就是使用生成二进制代码了，在生成二进制代码之前必须进行变量的解析，这里完成 class VariableProxy 到 class Variable 的绑定。

parser 需要在当前的 scope 里 declare 这个变量。对于一段 js 脚本，每个层次都对应一个 class Scope，该层次上的变量、作用域范围与此 scope 紧密相关。如脚本文件以及该文件里的函数是两个层次的 scope。

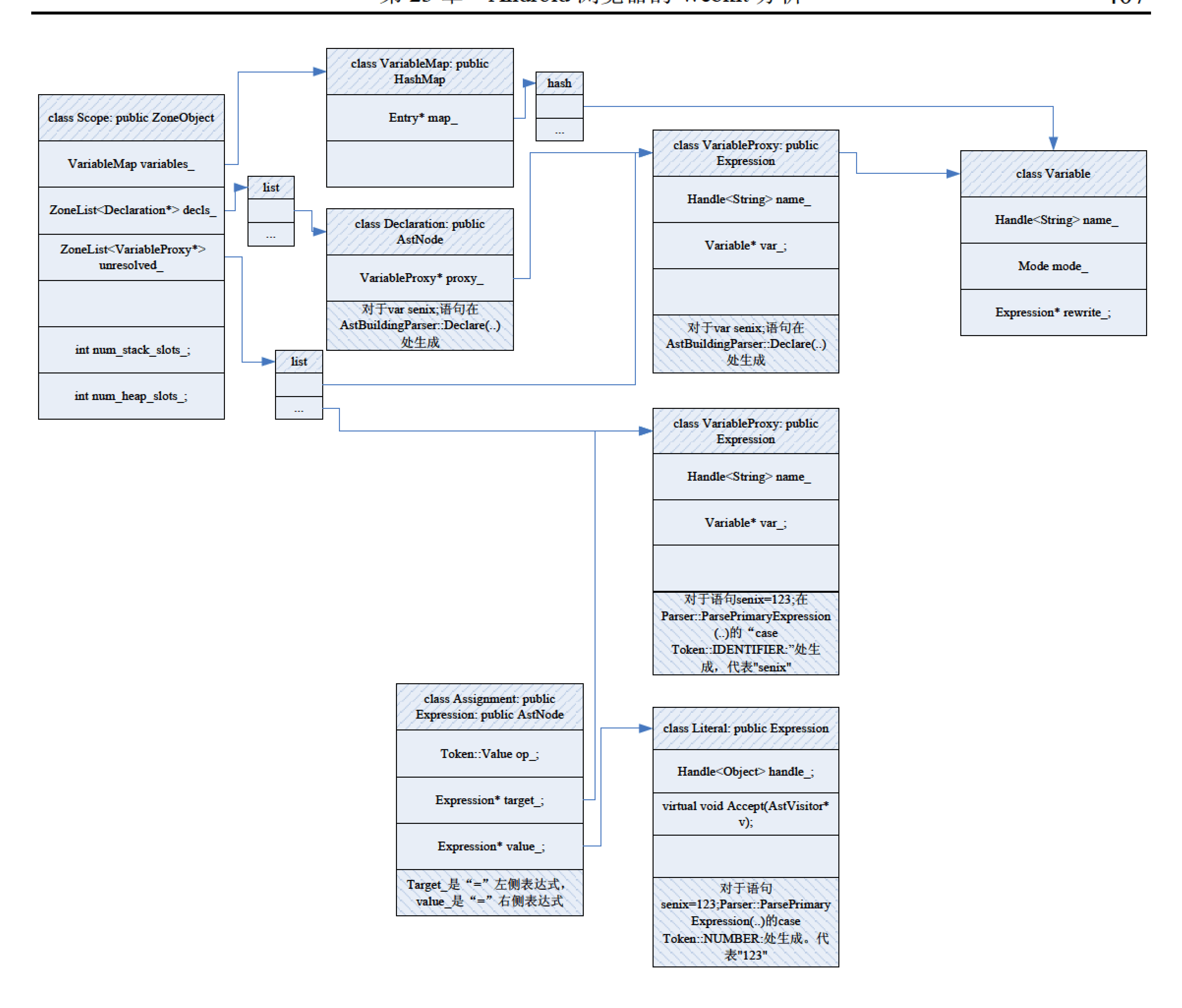


图 23-5 parser_scope_2 的结构

脚本文件的 scope 生成如下：

```
FunctionLiteral* Parser::ParseProgram(Handle<String> source,
                                      bool in_global_context) {


    /*根据参数 bool in_global_context 判定该 Scope 是何种类型，对于脚本文件级，
    in_global_context 为 1，该 Scope 的类型为 GLOBAL_SCOPE*/

  Scope::Type type =
    in_global_context
      ? Scope::GLOBAL_SCOPE
      : Scope::EVAL_SCOPE;
    …
//为该级的 js 代码创建 Scope，对于 js 文件级，top_scope_为 NULL
  Scope* scope = factory()->NewScope(top_scope_, type, inside_with());
    …
```

```
}
```

具体 Scope 创建根据 Parser 类型不同而异，对于 class AstBuildingParser，其创建函数如下：

```
Scope* AstBuildingParserFactory::NewScope(Scope* parent, Scope::Type type,
                                          bool inside_with) {
  Scope* result = new Scope(parent, type);
  result->Initialize(inside_with);
  return result;
}
```

23.2.3 语法分析的入口 Parser::ParseStatement(…)

在该函数里进行实质的语法分析。这是一个分拣器：分析出每个 js 语句的类型，然后针对不同的类型再进一步调用相关的分析函数。具体流程操作过程是，首先分离出有关键字的 statement。这些 statement 由关键字指明，有很清楚的含义，如 var forest;，接下来交给专门的 statement 分析器再进行更清楚的分析。比如 var 语句交给 Block* ParseVariableStatement(bool* ok);，if 语句交给 IfStatement* ParseIfStatement(ZoneStringList* labels, bool* ok);，交给而对于普通语句如 forest=1876+forest3;、函数调用等语句 Statement* ParseExpressionOrLabelledStatement(ZoneStringList* labels,bool* ok);做进一步分析。

源码解析如下，其分拣工作由一个 switch 进行。

```
Statement* Parser::ParseStatement(ZoneStringList* labels, bool* ok) {
  …
  switch (peek()) {
    …
    case Token::CONST:  // fall through
    case Token::VAR:
    //VAR 变量声明语句
      stmt = ParseVariableStatement(ok);
      break;
    …
    case Token::IF:
    //IF 语句
      stmt = ParseIfStatement(labels, ok);
      break;
    …
    case Token::TRY: {
    //try 语句
      …
      Block* result = NEW(Block(labels, 1, false));
      Target target(this, result);
      …
    }
```

```
    case Token::FUNCTION:
   //遇到一个函数
     return ParseFunctionDeclaration(ok);
    …
    default:
     //普通语句走到这里
      stmt = ParseExpressionOrLabelledStatement(labels, ok);
  }
…
}
```

23.2.4　普通语句的分析

接下来分析遇到具体的 JS 语法元素时的处理。

1. 变量声明

当遇到一个 var Token，parser 使用 Block* Parser::ParseVariableDeclarations(bool accept_IN, Expression** var, bool* ok) {…}来分析这个语句。

```
Block* Parser::ParseVariableDeclarations(bool accept_IN,
                                         Expression** var,
                                         bool* ok) {

  Variable::Mode mode = Variable::VAR;
  bool is_const = false;
  if (peek() == Token::VAR) {
    //将 parser 的扫描器指针向前推进到 Var 后面的位置
    Consume(Token::VAR);
    //CONST 语句与 VAR 处理相同
  } else if (peek() == Token::CONST) {
    //将 parser 的扫描器指针向前推进到 CONST 后面的位置
    Consume(Token::CONST);
    mode = Variable::CONST;
    is_const = true;
  } else {
    UNREACHABLE();  // by current callers
  }

  //生成一个 class Block: public BreakableStatement 节点
  Block* block = NEW(Block(NULL, 1, true));

  VariableProxy* last_var = NULL;  // the last variable declared
  int nvars = 0;  // the number of variables declared
```

```
  do {
    /*对于多个变量的声明语句，将 parser 的扫描器指针向前推进到变量声明的逗号后面*/
    if  (nvars  >  0)  Consume(Token::COMMA);       Handle<String>  name  =
ParseIdentifier(CHECK_OK);//提取出当前变量的标识符

//在当前 scope 里声明该变量，这是最重要的一步
    last_var = Declare(name, mode, NULL,
                            is_const /* always bound for CONST! */,
                            CHECK_OK);
    nvars++;

    Expression* value = NULL;
    int position = -1;

    //对于有初始值的声明，按赋值表达式处理
    if (peek() == Token::ASSIGN) {
      Expect(Token::ASSIGN, CHECK_OK);
      position = scanner().location().beg_pos;
      value = ParseAssignmentExpression(accept_IN, CHECK_OK);
    }

   …
  } while (peek() == Token::COMMA);
 …
  return block;
}
```

2. AssignmentExpression 表达式

对一个 AssignmentExpression 表达式的分析以 forest=99 为例，代码如下：

```
Expression* Parser::ParseAssignmentExpression(bool accept_IN, bool* ok) {
 /*对于“=”左边，执行 Parser::ParseConditionalExpression，该函数返回 “=”左侧
   的表达式，对于 forest=99;分析“=”左边的表达式*/
  Expression*    expression    =    ParseConditionalExpression(accept_IN,
CHECK_OK);
//如果接下来 token 不是“=”，直接返回该表达式
  if (!Token::IsAssignmentOp(peek())) {
    // Parsed conditional expression only (no assignment).
    return expression;
  }
…
  Token::Value op = Next();  // Get assignment operator.
```

```
//取出下一个操作符，对于 forest=99;，op 为“=”
  int pos = scanner().location().beg_pos;
//分析“=”右边的表达式
  Expression* right = ParseAssignmentExpression(accept_IN, CHECK_OK);

  …
/*左右两侧的语法结构都已分析完毕，根据生成的左右表达式和操作符 op 生成语句 iforest=99
的 statement*/
  return NEW(Assignment(op, expression, right, pos));

}
```

3. LogicalOrExpression '?' AssignmentExpression ':' AssignmentExpression 表达式

```
    /*该函数分析诸如 LogicalOrExpression '?' AssignmentExpression ':'
AssignmentExpression 的语句*/
Expression* Parser::ParseConditionalExpression(bool accept_IN, bool* ok) {
…
//分析 LogicalOrExpression 部分，即“? ”左侧
  Expression* expression = ParseBinaryExpression(4, accept_IN, CHECK_OK);

//如果该语句只有 LogicalOrExpression 部分，则 peek 不到“?”，返回
if (peek() != Token::CONDITIONAL) return expression;
//scanner 的指针移过“? ”
  Consume(Token::CONDITIONAL);
  …
//分析前一个 AssignmentExpression 部分即“? ”和“:”之间的表达式
  Expression* left = ParseAssignmentExpression(true, CHECK_OK);
//检查是否下面出现了“:”
  Expect(Token::COLON, CHECK_OK);
//分析后一个 AssignmentExpression 部分即“:”后面的表达式
  Expression* right = ParseAssignmentExpression(accept_IN, CHECK_OK);
  return NEW(Conditional(expression, left, right));
}
```

4. 用 ParseBinaryExpression(…)分析“?”左侧的表达式

二元函数分离器 Expression* Parser::ParseBinaryExpression(…);，看起来云里雾里，不得其解。其实这是因为其原函数里很多代码是用来优化 statement 的，先将优化部分去掉分析将其主干。

这个函数使用了一个重要工具：Expression* ParseUnaryExpression(…);，该函数暂且可以理解为在一串操作序列中取出第一个操作数，以 a+b+c 为例，Expression* ParseUnaryExpression(…);返回对应 a 的表达式。

再者操作符都有自己的优先级，int Precedence(Token::Value tok, bool accept_IN);这个函

数就是取出对应操作符的优先级（操作符优先级的定义参见 token.h）。

该函数最大的特点是递归分析，其每次递归下探的深度，取决由操作符的优先级。每次 Parser::ParseBinaryExpression(…)使用 Expression* ParseUnaryExpression(…),取出第一个操作数后，它会检查在取出的操作数后的操作符的优先级是否大于（高于）该操作数之前操作符的优先级。如果不是，就戛然而止，递归探底。这种情况下函数 Expression* Parser::ParseBinaryExpression(…)实际上就等效于 Expression* ParseUnaryExpression(…)，代码如下：

```
Expression* Parser::ParseBinaryExpression(int prec, bool accept_IN, bool*
ok) {
  ASSERT(prec >= 4);
//取出第一个操作数
  Expression* x = ParseUnaryExpression(CHECK_OK);
/*如果接下来操作符的优先级（Precedence(peek(), accept_IN)）低于第一个操作数前面操
  作符的优先级（int prec），就向上返回了*/

  for (int prec1 = Precedence(peek(), accept_IN); prec1 >= prec; prec1--){
//跑到这里说明遇到了更高优先级的操作符
    // prec1 >= 4
    while (Precedence(peek(), accept_IN) == prec1) {
//取出这个操作符
      Token::Value op = Next();
/*去取下一个操作数，参数 prec1 + 1，说明同一等级的操作符不会通过递归进行。而是用这个
  while 循环逐个分解*/
      Expression* y = ParseBinaryExpression(prec1 + 1, accept_IN, CHECK_OK);
//生成新的表达式
        x = NEW(BinaryOperation(op, x, y));
    }
  }
  return x;
}
```

再看完整的代码：

```
// Precedence >= 4
Expression* Parser::ParseBinaryExpression(int prec, bool accept_IN, bool*
ok) {
  ...
  //先进行一元表达式分析
  Expression* x = ParseUnaryExpression(CHECK_OK);
  for (int prec1 = Precedence(peek(), accept_IN); prec1 >= prec; prec1--)
{
    // prec1 >= 4
    while (Precedence(peek(), accept_IN) == prec1) {
```

```
    Token::Value op = Next();
    //逐层次分析二元表达式分析
    Expression* y = ParseBinaryExpression(prec1 + 1, accept_IN, CHECK_OK);
/*如果 x 和 y 都是数字就直接算出结果，合并表达式。不然后面的二进制生成器要对每个表达式都
  生成一遍代码*/
    if (x && x->AsLiteral() && x->AsLiteral()->handle()->IsNumber() &&
        y && y->AsLiteral() && y->AsLiteral()->handle()->IsNumber()) {
      double x_val = x->AsLiteral()->handle()->Number();
      double y_val = y->AsLiteral()->handle()->Number();

      switch (op) {
        case Token::ADD:
          x = NewNumberLiteral(x_val + y_val);
          continue;
        …
        case Token::SAR: {
          uint32_t shift = DoubleToInt32(y_val) & 0x1f;
          int value = ArithmeticShiftRight(DoubleToInt32(x_val), shift);
          x = NewNumberLiteral(value);
          continue;
        }
        default:
          break;
      }
    }

//如果 y 是数字且是除法操作，就直接算出结果，合并表达式

    // Convert constant divisions to multiplications for speed.
    if (op == Token::DIV &&
        y && y->AsLiteral() && y->AsLiteral()->handle()->IsNumber()) {
      double y_val = y->AsLiteral()->handle()->Number();
      int64_t y_int = static_cast<int64_t>(y_val);
      …
    }
    …
  }
 }
 return x;
}
```

5. 分析一元表达式

该函数的功能是：要么分离出一元表达式，要么分离出普通表达式，代码如下：

```
Expression* Parser::ParseUnaryExpression(bool* ok) {
   …
  //先取出操作符
  Token::Value op = peek();
//检查操作符是否为一元表达式操作符
  if (Token::IsUnaryOp(op)) {
    //取下一个元素
    op = Next();
    Expression* expression = ParseUnaryExpression(CHECK_OK);
    …
    if (expression != NULL && expression->AsLiteral() &&
        expression->AsLiteral()->handle()->IsNumber()) {
      …
      }
    }
    …
  } else if (Token::IsCountOp(op)) {
    …
    }
    …

  } else {
  //不是一元表达式
    return ParsePostfixExpression(ok);
  }
}
```

基础表达式分析以表示符为例如下：

```
Expression* Parser::ParsePrimaryExpression(bool* ok) {
  …
  switch (peek()) {
    …
    case Token::IDENTIFIER: {
     /*最基本的表示符都会跑到这里，对于 forest=99;，等号左侧的 forest 分析最终会走到
       这里*/
    //取出标识符将其放入 name 变量
      Handle<String> name = ParseIdentifier(CHECK_OK);

      if (is_pre_parsing_) {
        result = VariableProxySentinel::identifier_proxy();
      } else {
/*创建一个 class VariableProxy 类型的对象，并将其放入在当前的 scope 里的 unresolved_
  列表中*/
        result = top_scope_->NewUnresolved(name, inside_with());
      }
```

```
        break;
      }

    case Token::NUMBER: {
/*扫描器碰到了一个数字，对于 forest=99;，分析等号右侧的 99，最终会走到这里*/
      Consume(Token::NUMBER);
      //在 V8 内部是用一个双精度数来表示数字
      double value =
        StringToDouble(scanner_.literal_string(), ALLOW_HEX | ALLOW_
        OCTALS);

      result = NewNumberLiteral(value);
      break;
    }
    …
  }

  return result;
}
```

23.3　指 令 生 成

在 V8 parser 生成语法树之后，接下来是生成二进制指令。其基本做法是，按照语法树的基本结构，依次遍历其上的节点，然后根据每个节点的类型，找到其二进制指令的生成器。这些不同的节点类型有着其对应的 AstVisitor。

Class AstVisitor 位于 src/ast.h，对应于 AST NODE，对于不同的 AST NODE 有不同的 Visitor。

```
class AstVisitor BASE_EMBEDDED {
  void Visit(AstNode* node) { if (!CheckStackOverflow()) node-> Accept
  (this); }

#define DEF_VISIT(type)                          \
  virtual void Visit##type(type* node) = 0;
  AST_NODE_LIST(DEF_VISIT)
#undef DEF_VISIT
};

#define AST_NODE_LIST(V)                         \
  V(Declaration)                                \
  STATEMENT_NODE_LIST(V)                         \
  EXPRESSION_NODE_LIST(V)
```

可见 AST 节点列表包含了 statement 列表和表达式列表。

```
#define STATEMENT_NODE_LIST(V)                    \
  V(Block)                                        \
  …

#define EXPRESSION_NODE_LIST(V)                   \
  V(FunctionLiteral)                              \
  …
```

另一方面，对于每种架构的处理器，都会有自己的 class CodeGenerator，在 src/xxx/codegen-xxx.h：

```
class CodeGenerator: public AstVisitor{
…
/*该函数体现了二进制指令生成基本结构，这又是一个面向对象的方法：语法树体现脚本的逻辑结
  构和调用关系，其上语法节点对应着不同的 Visitor，其二进制指令的生成是由这些 Visitor
  具体完成的*/
void Visit(AstNode* node) { if (!CheckStackOverflow()) node-> Accept
(this); }
…
}
```

class CodeGenerator 里会实现 Class AstVisitor 里的 VisitXXXXX 函数，而所有表达式继承于 class RegExpTree，所有 statement 继承于 class AstNode，这两个类里都定义了虚函数 virtual void Accept(AstVisitor* v)；这些表达式和 statement 的 accept(…)函数的具体实现在 ast.cc 中：

```
#define DECL_ACCEPT(type)                                    \
  void type::Accept(AstVisitor* v) { v->Visit##type(this); }
AST_NODE_LIST(DECL_ACCEPT)
#undef DECL_ACCEPT
```

可见，AST_NODE_LIST 节点列表包含了 statement 列表和表达式列表，每种 STATEMENT_NODE 和 EXPRESSION_NODE 的 accept 函数为 v->Visit##type(this);。这样语法树被送到 CodeGenerator 后，针对不同类型的节点执行不同 AstVisitor 的 Vist 操作，即调用其 VisitXXXX 函数生成二进制指令。